НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра автоматизованих систем обробки інформації і управління

До захисту допущено:
В.о. завідувача кафедри
_____ *Олександр ПАВЛОВ*
(підпис) (вл.ім'я, прізвище)
"___"_____2020 р.

Дипломний проєкт
на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інформаційні управляючі системи та технології»
спеціальності 122 «Комп'ютерні науки та інформаційні технології»

на тему: «*Комплекс задач аналізу моделей для побудови Document Object Model*»

Виконав: студент *IV* курсу, групи *ІС-з61*

Загоровський Олександр Анатолійович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник *доц., к.ф.-м.н., доц. Гавриленко Олена Валеріївна* _____
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Консультант з норм контролю *доц., к.т.н, доц. Тєлишева Тамара Олексіївна* _____
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент *ст. викл. каф. ТК, к.т.н., Солдатова Марія Олександрівна* _____
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____
(підпис)

Київ – 2020 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет (інститут) *інформатики та обчислювальної техніки*
(повна назва)

Кафедра *автоматизованих систем обробки інформації і управління*
(повна назва)

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки та інформаційні технології»

Освітньо-професійна програма «Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ *Олександр ПАВЛОВ*
(підпис) (вл.ім’я, прізвище)
“___”_____2020 р.

З А В Д А Н Н Я
на дипломний проєкт студенту

Загоровському Олександру Анатолійовичу
(прізвище, ім’я, по батькові)

1. Тема проєкту «*Комплекс задач аналізу моделей для побудови Document Object Model*»
керівник проєкту *Гавриленко Олена Валеріївна, к.ф.-м.н., доцент*
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від “*7*”*травня* 2020 р. №1081-с
2. Термін подання студентом проєкту “*01*”*червня* 2020 року
3. Вихідні дані до проєкту
Технічне завдання

4. Зміст пояснювальної записки
1. Сучасні браузери та їх архітектура: високорівнева архітектура браузера, модуль відображення
2. Інформаційне забезпечення: DOM, Document API, віртуальна та інкрементальна моделі DOM
3. Побудова DOM за допомогою Javascript бібліотек: аналітичний огляд бібліотеки React, аналітичний огляд фреймворку Angular
4. Комплекс задач аналізу моделей побудови DOM: аналіз швидкості побудови та зміни DOM, аналіз використаної пам’яті при побудові DOM

5. Перелік графічного матеріалу

1. Схема структурна варіантів використань

2. Схема структурна станів моделей

3. Схема структурна класів програмного забезпечення

4. Схема структурна послідовності

5. Креслення вигляду екранних форм

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *«13» квітня 2020 року*

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	*Вивчення рекомендованої літератури*	*14.04.2020*	
2.	*Аналіз існуючих методів розв'язання задачі*	*17.04.2020*	
3.	*Постановка та формалізація задачі*	*20.04.2020*	
4.	*Розробка інформаційного забезпечення*	*23.04.2020*	
5.	*Алгоритмізація задачі*	*27.04.2020*	
6.	*Обґрунтування використовуваних технічних засобів*	*01.05.2020*	
7.	*Розробка програмного забезпечення*	*02.05.2020*	
8.	*Налагодження програми*	*08.05.2020*	
9.	*Виконання графічних документів*	*10.05.2020*	
10.	*Оформлення пояснювальної записки*	*11.05.2020*	
11.	*Подання ДП на попередній захист*	*15.05.2020*	
12.	*Подання ДП на основний захист*	*01.06.2020*	
13.	*Подання ДП рецензенту*	*02.06.2020*	

Студент Олександр ЗАГОРОВСЬКИЙ

Керівник Олена ГАВРИЛЕНКО

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проєкт	2	
2	А4	ДП 6109.00.000 ПЗ	Пояснювальна записка	62	
3	А4	ДП 6109.01.000 ТЗ	Технічне завдання	10	
4	А4	ДП 6109.02.000 ССВ	Схема структурна варіантів використань	1	
5	А4	ДП 6109.03.000 ССС	Схема структурна станів системи	1	
6	А4	ДП 6109.04.000 ССК	Схема структурна класів програмного	2	
			забезпечення		
7	А4	ДП 6109.05.000 ССП	Схема структурна послідовності	1	
8	А3	ДП 6109.06.000 КЕ	Креслення вигляду екранних форм	1	

Зм.	Арк.	ПІБ	Підп.	Дата	ДП 6109 00.000.00		
Розробн.		Загоровський О			Відомість дипломного проєкту	Літ.	Лист 1 / Листів 1
Керівн.		Гавриленко О.В.					
Консульт.						КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-з61	
Н/контр.		Тєлишева Т.О.					
В.о.зав.каф.		Павлов О.А.					

Пояснювальна записка
до дипломного проєкту

на тему: Комплекс задач аналізу моделей для побудови Document Object Model

Київ – 2020 року

АНОТАЦІЯ

Структура та обсяг роботи. Пояснювальна записка дипломного проєкту складається з чотирьох розділів, 24 малюнків, 5 таблиць, 3 додатки, 16 джерел.

Дипломний проєкт присвячений розробці комплексу задач аналізу моделей побудови об'єктної моделі документа браузера.

Мета роботи полягає у аналізі існуючих моделей побудови та зміни DOM, розробці реалізацій віртуальної та інкрементальної моделей, порівнянні швидкості виконання та кількості використаної пам'яті для кожної з моделей.

У першому розділі було розглянуто високорівневу архітектуру сучасних браузерів та модулю відображення браузера.

У другому розділі було розглянуто віртуальну та інкрементальну моделі побудови і зміни DOM за допомогою Document API.

У третьому розділі було досліджено архітектуру та модель побудови і зміни DOM JavaScript бібліотеками React та Angular.

У четвертому розділі було виконано порівняльний аналіз методів та моделей побудови і зміни DOM за допомогою обраних JavaScript бібліотек та власних імплементацій. Було порівняно час та кількість використаної пам'яті за допомогою вбудованих інструментів браузера.

ОБ'ЄКТНА МОДЕЛЬ ДОКУМЕНТА, DOM, JAVASCRIPT, REACT, ANGULAR, VUE.

					ДП 6109.00.000 ПЗ		
		Прізвище	Підпис	Дата			
Розроб.		Загоровський О			Комплекс задач аналізу моделей для побудови	Літ.	Лист 2 / Листів
Перевірив.		Гавриленко О.В.					
						КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-з61	
Н. кон.		Тєлишева Т.О.					
Затв.		Павлов О.А.					

ABSTRACT

Structure and scope of work. The explanatory note of the diploma project consists of four sections, 24 figures, 5 tables, 3 appendices, 16 sources.

The diploma project is devoted to development of a complex of problems of the analysis of models of construction of object model of the browser document.

The purpose of the work is to analyze the existing models of building and modifying the DOM, developing implementations of virtual and incremental models, comparing the execution speed and the amount of memory used for each of the models.

The first section discusses the high-level architecture of modern browsers and the browser display module.

The second section discusses the virtual and incremental models of building and modifying a DOM using the Document API.

The third section examines the architecture and model of building and modifying DOM JavaScript by the React and Angular libraries.

In the fourth section, a comparative analysis of methods and models of building and modifying the DOM using selected JavaScript libraries and native implementations. The time and amount of memory used were compared using the browser's built-in tools.

DOCUMENT OBJECT MODEL, DOM, JAVASCRIPT, REACT, ANGULAR, VUE.

ЗМІСТ

ВСТУП

Класична архітектура web-додатків, в якій побудова HTML документа відбувається на сервері була дуже популярна довгі роки, але з плином часу користувачі інтернету та конкуренція на ринку web-додатків змушували розробників створювати все більші та все складніші додатки. З часом стало зрозуміло що така архітектура не є оптимальною, а деякий функції web-додатків і зовсім неможливо реалізувати в її рамках, тому вона потребувала змін. Сьогодні, в складних web-додатках, більшість функціоналу реалізована на клієнтському рівні архітектури, тому нерідко, сервер є лише адаптером між клієнтом та базою даних. Так, в цілому архітектура додатків все ще є типу клієнт-сервер, але архітектура саме клієнта значно еволюціонувала та стала набагато складнішою. Все це призвело до появи концепції односторінкових додатків в яких клієнт містить більшість логіки, може мати власне сховище, та навіть власну маршрутизацію, що відрізняється від серверної. В свою чергу це призвело до розвитку браузерів, а саме до розвитку інтерпретатора JavaScript, та ядра модуля відображення браузера.

На сьогодні побудова та внесення змін до об'єктної моделі документа(Document Object Model, далі DOM) браузера є основою розробки сучасних односторінкових додатків у web. Це є однією з найресурсозатратніших операцією в браузері, тому нерідко не оптимальна модель побудови або зміни DOM може стати проблемою, результатом якої можуть бути візуальні дефекти додатку.

Дана дипломна робота присвячується аналізу моделей побудови та зміни DOM, та вирішенню комплексу задач для аналізу цих моделей.

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Опис предметного середовища

Головним предметним середовищем даної роботи є браузер.

Браузер - програма, призначена для перегляду веб-сайтів, яка за фактом інтерпретує HTML, CSS та JavaScript код в зрозуміле для користувача візуальне представлення.

1.1.1 Високорівнева архітектура браузера

Нижче перераховані основні компоненти браузера.

1. Інтерфейс - включає адресний рядок, кнопки навігації, меню закладок і т. п. До нього включають усі елементи, крім головного вікна, в якому відображається HTML сторінка.
2. Рушій браузера - відповідає за взаємодію між інтерфейсом та модулем відображення.
3. Модуль відображення - відповідає за виведення запитаного вмісту на екран. Наприклад, коли завантажується HTML-файл, модуль відображення виконує синтаксичний аналіз коду HTML та CSS, і виводить результат на екран.
4. Мережеві компоненти - призначені для виконання мережевих запитів, наприклад, HTTP-запити. Їхній інтерфейс не залежить від типу платформи, для кожної є власна реалізація.
5. Виконавча частина користувацького інтерфейсу - використовується для відображення основних компонентів, таких як поля зі списками або вікна. Її інтерфейс є універсальним та не залежить від типу платформи виконання. Виконавча частина завжди використовує методи інтерфейсу операційної системи користувача.
6. Інтерпретатор JavaScript - виконує синтаксичний аналіз та інтерпретацію коду JavaScript.

7. Сховище даних - необхідно для зберігання даних. Браузер зберігає на жорсткий диск дані різних типів, наприклад дані з LocalStorage. У специфікації HTML5 є визначений термін "Web-SQL": це майже повноцінна база даних у браузері.

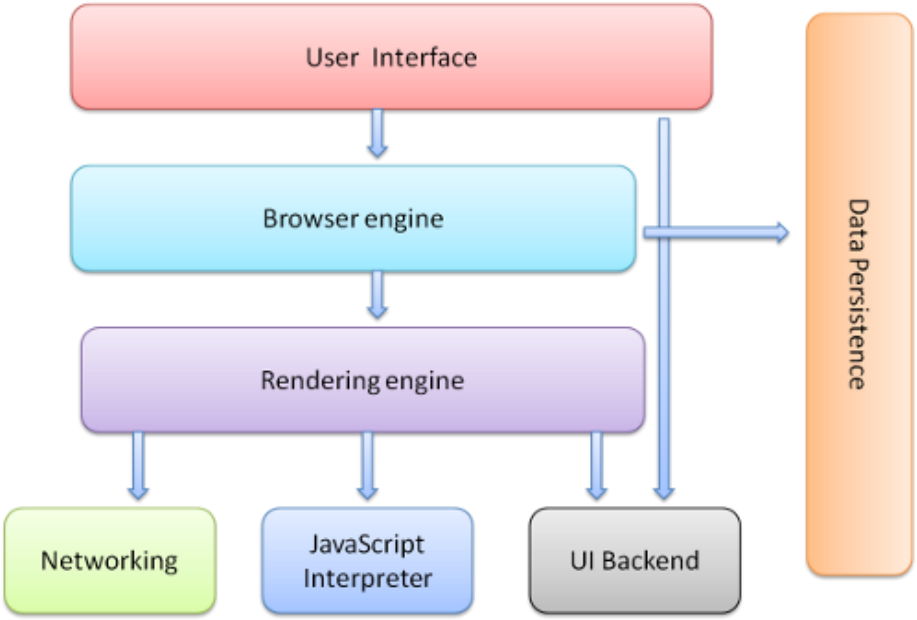


Рисунок 1.1 - Основні компоненти браузера

1.1.2 Модуль відображення браузера

Як зазначено у роботі [7], модуль відображення відповідає за виведення запитаного вмісту на екрані користувача.

Початково він здатний відтворювати XML або HTML документи, а також картинки. Спеціальні модулі, що є розширеннями для браузерів, дозволяють відображувати файли іншого змісту, наприклад PDF-файли.

У браузерах Firefox, Chrome і Safari використовуються два модуля відображення. У Firefox застосовується Gecko - власна розробка компанії Mozilla, а в Safari і Chrome використовується WebKit.

WebKit є модуль відображення з відкритим вихідним кодом, який був спочатку розроблений для платформи Linux і адаптований компанією Apple для Mac OS і Windows.

1.1.2.1 Схема роботи модуля відображення

Модуль відображення отримує зміст запитаного документа по протоколу мережевого рівня, це зазначено у роботах [3] та [6]. Схема роботи модуля відображення зображена на рисунку 1.2.

Рисунок 1.2 - Схема роботи модуля відображення

Після виконання синтаксичного аналізу HTML-документа модуль відображення переводить теги в вузли DOM. Інформація про стилі сторінки зчитується як із зовнішніх CSS-файлів, так і з HTML елементів <style>. Ці інструкції та дані по відображенню HTML-файла використовуються для створення ще додаткового дерева - дерева відображення.

Воно складається з елементів з візуальними атрибутами, такими як колір та розмір. Елементи розташовуються один за одним, в порядку, в якому вони повинні бути виведені на екран.

Після побудови дерева відображення відбувається компонування елементів, в ході якого кожному елементу присвоюються координати x та y точки на екрані, де він повинен бути відображений. Потім виконується відображення, при якому вузли дерева відображення послідовно відображаються за допомогою виконавчої частини призначеного для користувача інтерфейсу.

Для підвищення швидкості роботи модуль відображення намагається вивести зміст на екран якомога раніше, тому побудова дерева відображення і компонування можуть початися ще у процесі синтаксичного аналізу HTML коду. Одні частини документа передаються по мережі, в той час як інші аналізуються і виводяться на екран.

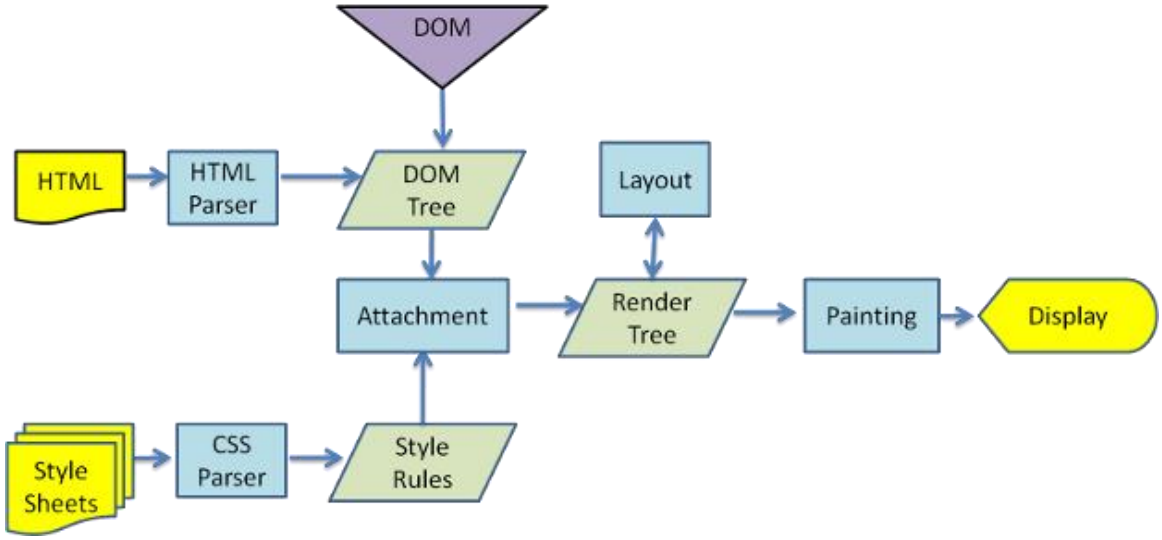


Рисунок 1.3 - Схема роботи модуля відображення WebKit

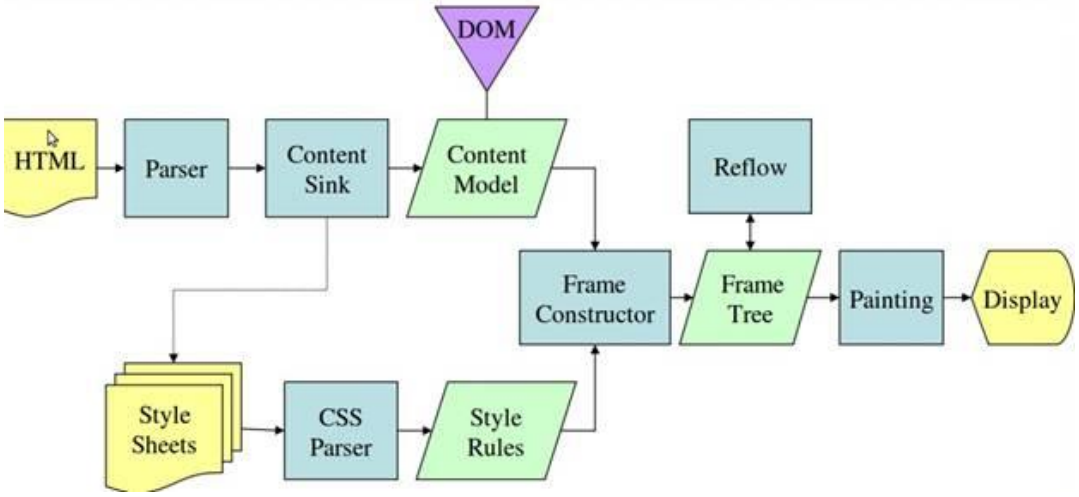


Рисунок 1.4 - Схема роботи модуля відображення Gecko

Як видно з рисунків 1.3 та 1.4, в WebKit і Gecko використовується різна термінологія, але принципи їх роботи практично ідентичні.

Дерево візуально відформатованих елементів моделі Gecko називається деревом каркасів - frame tree, де кожен елемент є каркасом. В моделі WebKit аналогічним є дерево відображення - render tree, яке складається з об'єктів відображення - render objects. Розміщення елементів в моделі WebKit називається процесом компонування (layout), в Gecko - поповненням (reflow). Об'єднання елементів DOM та візуальних особливостей для створення дерева відображення в моделі WebKit називається вкладенням (attachment). Невелика відмінність моделі Gecko від WebKit, полягає в тому, що між завантаженим

HTML-файлом та побудованим деревом DOM є ще один рівень - буфер контенту (content sink), що служить для створення елементів DOM.

Як зазначено у роботі [5], що Chrome, використовує декілька копій модуля відображення, по одному на кожну сторінку браузера, що оброблюються в окремих процесах.

1.1.2.2 Синтаксичний і лексичний аналізатори

Під синтаксичним аналізом документа мається на увазі його перетворення в придатну для читання і виконання структуру. Результатом синтаксичного аналізу, як правило, є дерево вузлів, що представляють структуру документа. Воно називається деревом синтаксичного аналізу, або просто синтаксичним деревом.

Наприклад, в результаті синтаксичного аналізу виразу 2 + 3 - 1 може бути сформоване наступне дерево, що зображене на рисунку 1.5.

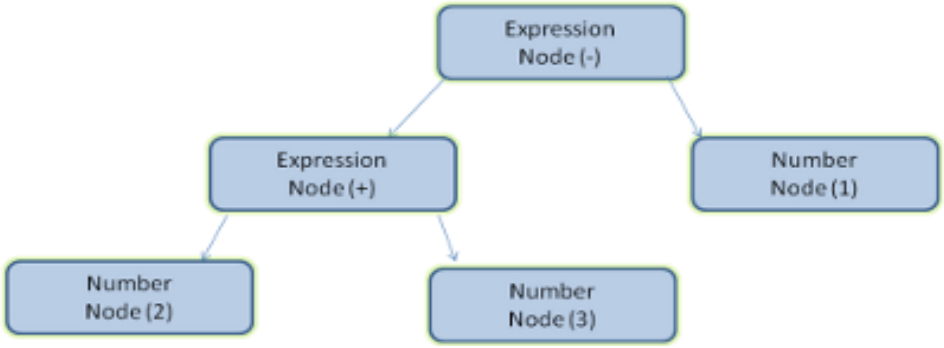


Рисунок 1.5 - Синтаксичне дерево для математичного виразу 2 + 3 – 1

У роботі [6] зазначено, що синтаксичний аналіз працює на основі певних правил, які визначаються мовою та форматом документа.

Для кожного формату існують граматичні правила, що складаються зі словника і синтаксису. Вони утворюють безконтекстну граматику. Природні мови не підкоряються правилам безконтекстної граматики, тому стандартні техніки синтаксичного аналізу для них не підходять.

Разом з синтаксичним застосовується лексичний аналіз.

Лексичний аналіз являє собою поділ інформації на токени, або лексеми. Токени утворюють словник тієї чи іншої мови і є конструктивними

елементами для створення документів. У природній мові токенами б були всі слова, які можна знайти в словниках.

Аналіз документа зазвичай виконується двома компонентами: лексичним аналізатором, що розбиває вхідну послідовність символів на дійсні токени, і синтаксичним аналізатором, що аналізує структуру документа згідно синтаксичних правил цієї мови і формує синтаксичне дерево. Аналізатори ігнорують неінформативні символи, такі як прогалини і переноси рядків.

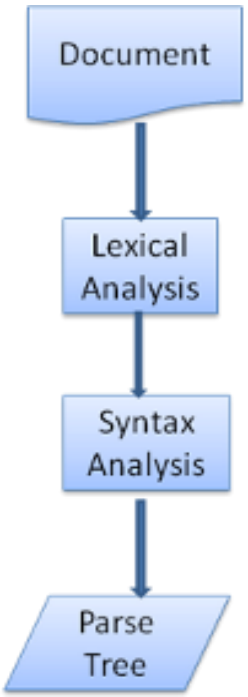


Рисунок 1.6 - Перехід від вихідного документа до синтаксичного дерева

Синтаксичний аналіз є ітеративним процесом тому синтаксичний аналізатор зазвичай запитує у лексичного новий токен і перевіряє його на предмет відповідності якому-небудь з синтаксичних правил. Якщо вдається встановити відповідність, для токена створюється новий вузол в синтаксичному дереві, а аналізатор запитує наступний токен.

Якщо токен не відповідає жодному правилу, синтаксичний аналізатор відкладає його і запитує наступні маркери. Цей процес відбувається, поки не буде знайдено правило, якому б відповідали всі відкладені токени. Якщо знайти таке правило не вдається, аналізатор створює виняток. Це означає, що

документ містить синтаксичні помилки і не може бути опрацьований повністю.

Синтаксичне дерево не завжди буває остаточним результатом. Синтаксичний аналіз часто використовується в процесі перекладу вхідного документа в потрібний формат. Прикладом може служити компіляція. Компілятор, який переводить вихідний код в машинний, спочатку розбирає його і формує синтаксичне дерево, а лише потім створює на основі цього дерева документ з машинним кодом.

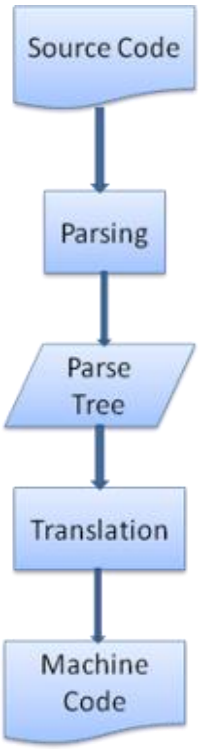


Рисунок 1.7 - Етапи компіляції

На рисунку 1.5 зображено синтаксичне дерево, побудоване на основі математичного виразу. Визначимо елементарну математичний мову і розглянемо процес синтаксичного аналізу.

Словник нашої мови може містити цілі числа, знаки "плюс" і "мінус".

Синтаксис:

- структурними елементами мови є вираження, операнди і оператори;
- мова може містити будь-яку кількість виразів;

- вираз - це послідовність, що складається з операнда, оператора і ще одного операнда;
- оператор - це токен "плюс" або "мінус";
- операнд - це токен цілого числа або вираз.

Розглянемо вхідну послідовність символів 2 + 3 - 1.

Перший елемент, який відповідає правилу, це 2 (згідно з правилом вище, це операнд). Другий такий елемент це 2 + 3 (послідовність, що складається з операнда, оператора і ще одного операнда, визначена правилом вище). Наступне відповідність ми знайдемо в самому кінці: послідовність 2 + 3 - 1 є виразом. Так як 2 + 3 - це операнд, ми отримуємо послідовність, що складається з операнда, оператора і ще одного операнда, що відповідає визначенню вираження. Рядок 2 + + не містить відповідностей правилам, тому був би розцінений як недійсний.

Формальне визначення словника і синтаксису.

Словник зазвичай складається з регулярних виразів. Мову з прикладу вище можна було б визначити так:

INTEGER: 0 | [1-9] [0-9] *

PLUS: +

MINUS: -

Типи синтаксичних аналізаторів.

Синтаксичні аналізатори бувають двох типів: низхідні і висхідні. Перші виконують аналіз зверху вниз, а другі - від низу до верху. Спадні аналізатори розбирають структуру верхнього рівня і шукають відповідності синтаксичним правилам. Висхідні аналізатори спочатку обробляють вхідну послідовність символів і поступово виявляють в ній синтаксичні правила, починаючи з правил нижнього і закінчуючи правилами верхнього рівня.

Розглянемо, як ці два типи аналізаторів впоралися б з нашим прикладом.

Спадний аналізатор почав би з правила верхнього рівня і визначив би, що 2 + 3 - це вираз. Потім він визначив би, що 2 + 3 - 1 також є виразом (в

процесі визначення виразів виявляються і відповідності іншим правилам, проте першим завжди розглядається правило верхнього рівня). Висхідний аналізатор обробляв би послідовність символів, поки не знайшов би відповідне правило, яким можна замінити виявлений фрагмент, і так до кінця послідовності. Вирази з частковим відповідністю при цьому поміщаються в стек аналізатора. При роботі такого аналізатора вхідна послідовність символів зсувається вправо (уявіть курсор, який поміщений в початок послідовності і в ході аналізу зсувається вправо) і поступово зводиться до синтаксичних правил.

Автоматичне створення синтаксичних аналізаторів.

Існують спеціальні програми для створення синтаксичних аналізаторів, які називаються генераторами. Досить завантажити в генератор граматику мови (словниковий запас і синтаксичні правила), і він автоматично створить аналізатор. Для створення синтаксичного аналізатора необхідно глибоке розуміння принципів його роботи, тому генератори бувають вельми корисні.

У WebKit використовується два відомих генератора: Flex для створення лексичного і Bison для створення синтаксичного аналізатора (вони також зустрічаються під назвами Lex і Yacc). У Flex завантажується файл з визначеннями токенів в регулярних виразах, а в Bison - синтаксичні правила мови в форматі BNF.

1.2 Огляд існуючих рішень

Модель та методи роботи з DOM до сьогодні не стандартизовані. Вони є лише набором рекомендацій від розробників браузерів тому одні з найвідоміших компанії світу, такі як Google, Microsoft, Facebook, розробляють свої рішення даної проблеми.

Усі існуючі рішення імплементують дві основні моделі побудови та зміни DOM:

– віртуальна;

– інкрементальна.

На основі цього мною було обрано кілька найбільш поширених та найпрогресивніших реалізацій, що мають підтримку великих компаній, використовуються тисячами розробників, мають широкий API та використовуються для вирішення різних в залежності від типу браузера(браузери мобільних пристроїв або настільних комп'ютерів) задач.

Існуючими рішеннями є:

- фреймворк Angular що розробляється компанією Microsoft, використовує інкрементальну модель;
- бібліотека React, що розробляється компанією Facebook, використовує віртуальну модель.

Висновок до розділу

В даному розділі було розглянуто роботу модуля відображення браузера. Роботу синтаксичного і лексичного аналізаторів. Описано процес побудови відображення від початку аналізу HTML сторінки до побудови DOM в рушіях WebKit і Gecko.

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 DOM

DOM - об'єктна модель документа (Document Object Model) - служить для представлення HTML-документа як ієрархії об'єктів, які реалізують певні інтерфейси. Специфікацію DOM розробляє консорціум W3C.

У корені дерева знаходиться об'єкт Document, інтерфейс якого містить фабричні методи, необхідні для створення дочірніх об'єктів, які в свою чергу володіють додатковими атрибутами для внутрішнього використання. Усі створені об'єкти мають атрибут ownerDocument, який пов'язує їх із документом, у контексті якого вони були створені. Деякі типи об'єктів можуть мати дочірні об'єкти різних типів, деякі, лише певних типів, а деякі, зовсім не можуть мати дочірніх об'єктів. Типи об'єктів:

– Document - кореневий елемент, може бути лише один на сторінці;

– DocumentFragment;

– DocumentType;

– EntityReference;

– Element;

– Attr;

– ProcessingInstruction - не може мати дочірні об'єктів;

– Comment - не може мати дочірні об'єктів;

– Text - не може мати дочірні вузлів;

– CDATASection - не може мати дочірні об'єктів;

– Entity;

– Notation - не може мати дочірні об'єктів.

Розглянемо приклад HTML розмітки:

```
<html>
<body>
    <p>
```

Paragraph

</ p>

 <img /> </ div>

</ body>

</ html>

Дерево DOM для цієї розмітки зображено на рисунку 2.1.

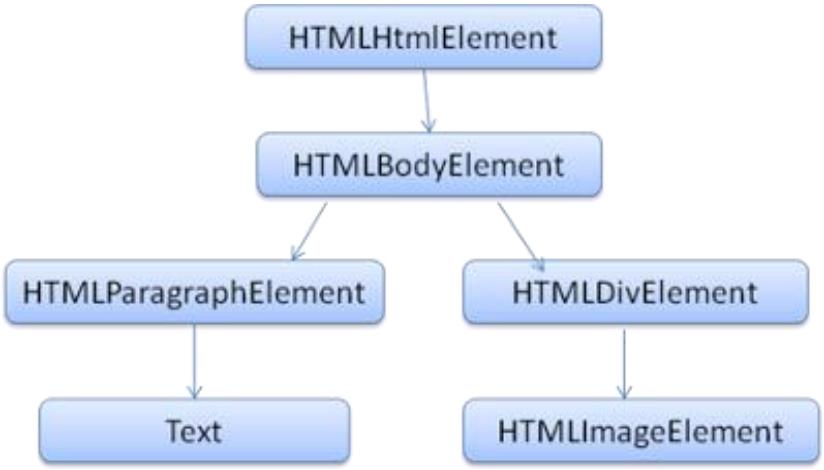


Рисунок 2.1 - Дерево DOM

2.1.1 Синтаксичний та лексичний аналіз

2.1.1.1 Алгоритми синтаксичного аналізу

Як зазначено у роботі [9], синтаксичний аналіз HTML коду неможливо виконати за допомогою стандартних аналізаторів.

Причини цього:

- мова має "не строгий" характер;
- у браузерах реалізовано механізми обробки типових помилок у коді HTML;
- алгоритм синтаксичного аналізу HTML характеризується особливість повторного входження тегів. Вихідний файл зазвичай не змінюється у процесі аналізу, однак якщо HTML теги script, містять виклики функції document.write, які в свою чергу можуть додавати нові елементи, вхідний код може змінюватися.

В основі алгоритму синтаксичного аналізу є два етапи: лексичний аналіз та побудова дерева. В процесі лексичного аналізу послідовність символів файлу розбивається на токени. До токенів коду HTML відносяться елементи що відкриваються, закриваються, а також назви і значення особливостей елементів. Лексичний аналізатор знаходить токен, надає його конструктору дерев і переходить до наступного символу в пошуку наступного токена, так відбувається до закінчення вхідної послідовності символів. Цей процес зображено на рисунку 2.2.

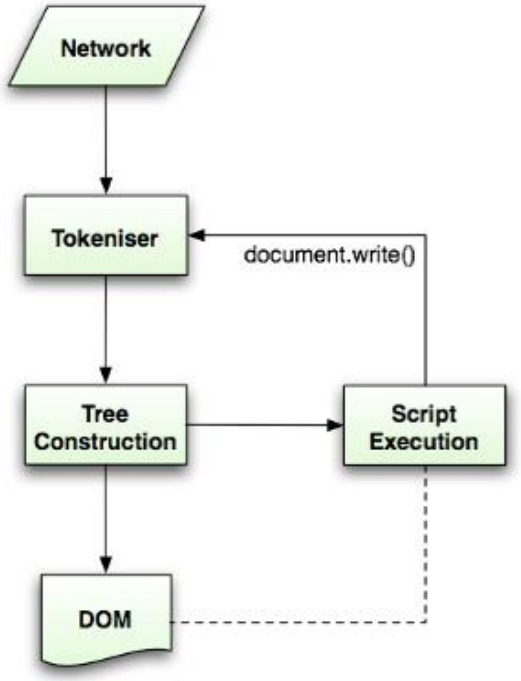


Рисунок 2.2 - Етапи синтаксичного аналізу HTML

2.1.1.2 Алгоритм лексичного аналізу

Результатом роботи алгоритму є HTML токен. Алгоритм виражений у вигляді автомата з кінцевим числом станів. У кожному стані обробляється один або кілька символів вхідної послідовності, на основі яких визначається наступний стан. Стан залежить від етапу лексичного аналізу та етапу формування дерева, тобто обробка одного і того ж символу може призвести до різних результатів (різних станів) в залежності від поточного стану.

Розглянемо спрощений приклад, який допоможе нам краще зрозуміти принцип його роботи.

Виконаємо лексичний аналіз простого коду HTML:

<html>

<body>

Hello world

</ body>

</ html>

Початковий стан - "дані". Коли аналізатор виявляє символ відкриття тегу <, стан змінюється на "відкритий тег". Якщо далі виявляється буква (a-z), створюється токен відкритого тегу, а стан змінюється на "назва тегу". Воно зберігається, поки не буде виявлений символ закриття тегу >. Символи по одному додаються до назви нового токена. У нашому випадку виходить токен html.

При виявленні символу > токен вважається готовим і аналізатор повертається в стан "дані". Тег <body> обробляється точно так же. Таким чином, аналізатор вже згенерував теги html і body і повернувся в стан "дані". Виявлення літери H у фразі Hello world веде до генерації токена символу. Те ж відбувається з іншими буквами, поки аналізатор не дійде до символу < в тезі </ body>. Для кожного символу фрази Hello world створюється свій токен.

Далі аналізатор знову повертається в стан "відкритий тег". Виявлення символу / веде до створення токена закритого тегу і переходу в стан "назва тега". Воно зберігається, поки не буде виявлений символ >. У цей момент генерується токен нового тега, а аналізатор знову повертається в стан "дані". Послідовність символів </ html> обробляється, як описано вище.

Процес лексичного аналізу зображено на рисунку 2.3.

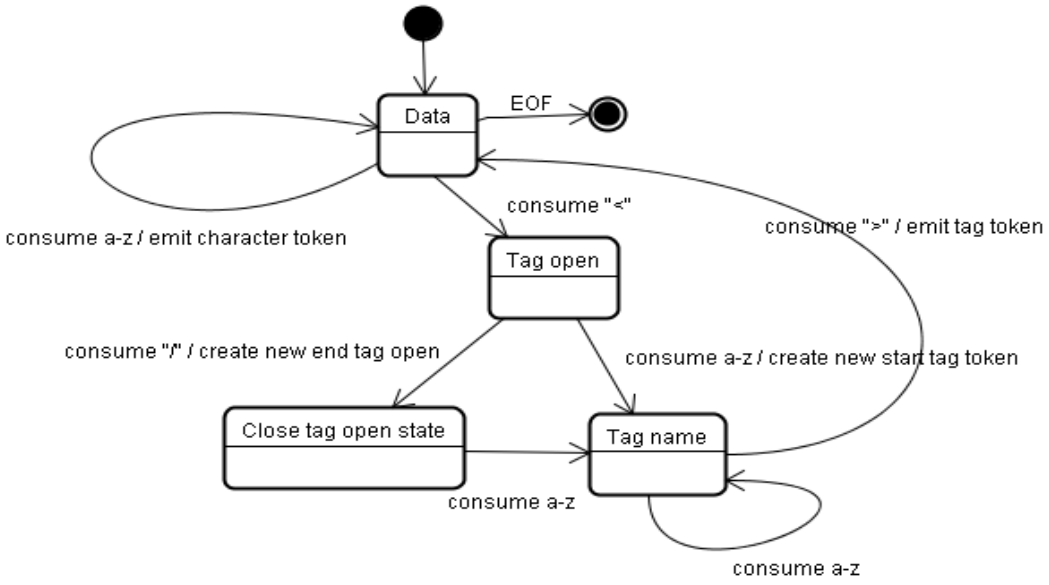


Рисунок 2.3 - Лексичний аналіз вхідної послідовності символів

2.1.1.3 Синтаксичний аналіз CSS

На відміну від HTML, в CSS використовується безконтекстна граматика, тому для аналізу підходять стандартні засоби. Крім того, лексичні і синтаксичні правила CSS визначені в специфікації CSS.

Як зазначено у роботі [11], лексична граматика (словник) визначається регулярними виразами для кожного токена:

- nmstart - [_a-z]|{nonascii}|{escape};
- comment - \/*[^*]**+([^/*][^*]**+)*\/;
- nonascii - [\200-\377];
- name - {nmchar}+;
- nmchar - [_a-z0-9-]|{nonascii}|{escape};
- ident - {nmstart}{nmchar}*;
- num - [0-9]+|[0-9]*"."[0-9]+.

Ident - це ідентифікатор, який використовується як назва класу.

Name - це елемент id, для посилання на нього використовується символ решітки (#). Синтаксичні правила аналізу CSS описані в форматі BNF.

ruleset

: simple_selectors [combinator selectors | S+ [combinators? selectors]?]?;
'{' S* declarations [';' S* declarations]* '}' S*
| [HASH | classes | attribs | pseudos]+;
: Selectors [',' S * selectors] *
simple_selectors
: '.' IDENT;
selectors
| [HASH | classes | attribs | pseudos]+;
: element_names [HASH | classes | attribs | pseudos]*;

Набір правил (ruleset) являє собою описану нижче структуру.

```
div.active,
a:visited {
    background: black;
    text-align: center;
}
```

Елементи div.active і a:visited - це селектори. Діючі правила даного набору укладені у фігурні дужки. Формально ця структура визначається так:

ruleset
: simple_selectors [combinator selectors | S+ [combinators? selectors]?]?;
: Selectors [',' S * selectors] *;

Це означає, що набір правил діє як селектор або як кілька селекторів, розділених комами і пробілами (S означає пробіл). Набір правил містить одне або кілька оголошень, між якими ставиться крапка з комою. Вони укладені у фігурні дужки.

Синтаксичний аналізатор CSS в WebKit.

У WebKit для автоматичного створення синтаксичних аналізаторів CSS використовуються генератори Flex і Bison. Bison служить для створення висхідних аналізаторів, при роботі яких вхідна послідовність символів

зсувається вправо. У Firefox використовується спадний аналізатор, розроблений організацією Mozilla. В обох випадках файл CSS розбирається на об'єкти StyleSheet, що містять правила CSS. Об'єкт правил CSS містить селектор і оголошення, а також інші об'єкти, характерні для граматики CSS (рисунок 2.4).

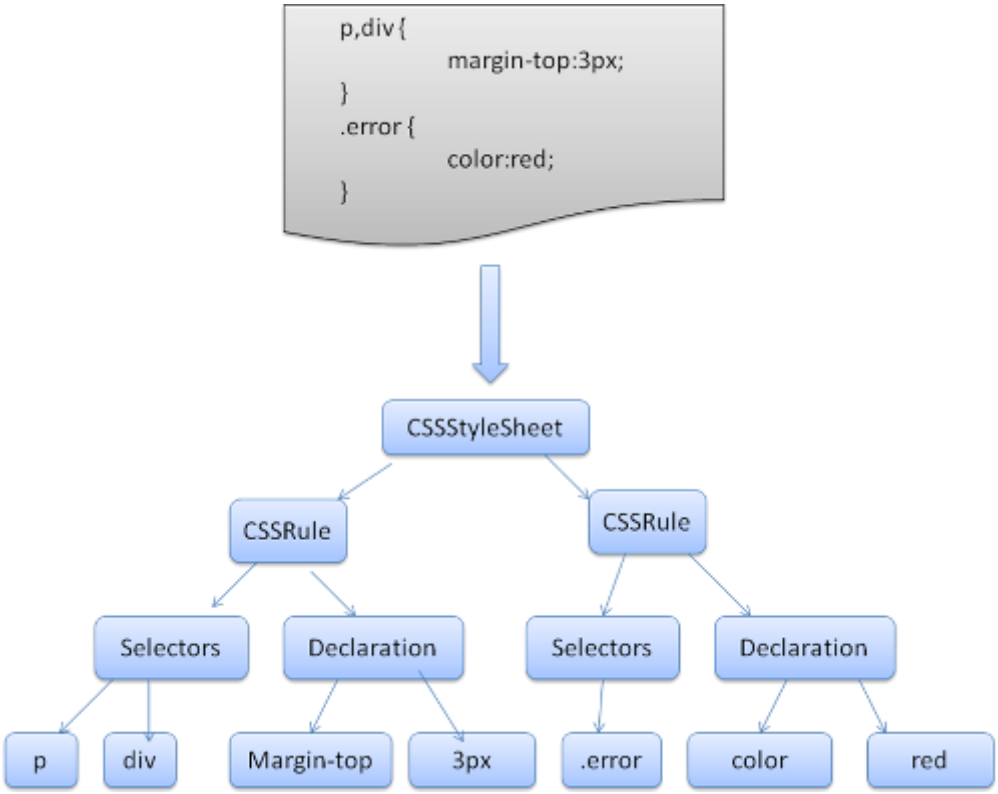


Рисунок 2.4 - Синтаксичний аналіз CSS

2.1.1.4 Обробка синтаксичних помилок

На сторінці HTML ви ніколи не побачите помилку "Неприпустимий синтаксис". Браузери вміють коригувати помилки змісту, не перериваючи роботу.

Розглянемо наступний код HTML:

```
<html>
    <mytag>
    </ mytag>
    <div>
    <p>
```

```
</ div>
Really lousy HTML
</ p>
</ html>
```

У цьому короткому фрагменті порушено безліч правил (mytag не є стандартним тегом, теги p та div вкладені невірно і т. Д.), Однак браузер відображає зміст коректно. Велика частина коду синтаксичного аналізатора служить для виправлення помилок розробників.

У браузерах використовуються дуже схожі механізми обробки помилок, але вони не описані в поточній специфікації HTML. Як і закладки або кнопки навігації, вони просто з'явилися в результаті багаторічної еволюції браузерів. Існують неприпустимі конструкції HTML, які досить часто зустрічаються на сайтах, і різні браузери виправляють їх схожими способами.

У специфікації HTML5 визначені деякі вимоги до цього механізму. У WebKit вони вказані в коментарі до класу parser.

Після синтаксичного аналізу браузер позначає документ як інтерактивний і починає аналіз відкладених скриптів, які необхідно виконати після завершення аналізу документа. Стан документа потім змінюється на "готовий", і викликається подія load.

Синтаксичний аналізатор обробляє вхідні токени і створює дерево документа. Якщо документ написаний без помилок, виконується його стандартний аналіз.

На жаль, багато документів HTML містять помилки, і синтаксичний аналізатор повинен бути готовий до них. Аналізатор повинен вміти обробляти як мінімум перераховані нижче типи помилок:

– Використання елементу, що додається явно заборонено одним із зовнішніх тегів. В цьому випадку необхідно закрити всі теги, крім того, який забороняє використання даного елементу, і додати цей елемент в самому кінці;

- Елемент не можна додати безпосередньо. Можливо, автор документу забув вставити тег між елементами (або такий тег необов'язковий). Це стосується тегів HTML, HEAD, BODY, TBODY, TR, TD, LI;
- Блоковий елемент доданий всередину рядкового. Необхідно закрити всі рядкові елементи аж до наступного в ієрархії блокового елемента;
- Якщо це не допомагає, необхідно закривати елементи, поки не з'явиться можливість додати потрібний елемент або проігнорувати тег.

Прикладом того, як WebKit обробляє деякі помилки може бути тег </br> замість <br>. На деяких сайтах можна зустріти тег </br> там, де повинен бути <br>. Щоб відобразити вміст в браузерах IE і Firefox, WebKit обробляє цей тег як <br>.

Код синтаксичного аналізатора, зазначений у роботі [8]:

```
if (t->isCloseTag(brTag) && m_document->inCompatMode()) {
    reportError(MalformedBRError);
    t->beginTag = true;
}
```

2.1.2 Алгоритм побудови дерева DOM

При створенні синтаксичного аналізатора формується об'єкт Document. На етапі побудови дерева DOM, в корені якого знаходиться цей об'єкт, Document змінюється і до нього додаються нові елементи.

Кожен вузол, що генерується лексичним аналізатором, обробляється конструктором дерев. Для кожного токена створюється свій елемент DOM за певний специфікацією.

Елементи додаються не тільки в дерево DOM, а й в стек відкритих елементів, який служить для виправлення неправильно вкладених або незакритих тегів. Алгоритм також виражається у вигляді автомата з кінцевим числом станів, які називаються "способами включення" (insertion mode).

Розглянемо етапи створення дерева для наступного фрагмента коду:

```
<html>
    <body>
    Hello world
    </ body>
</ html>
```

На початку етапу побудови дерева у нас є послідовність токенів, отримана в результаті лексичного аналізу. Перший стан називається вихідним. При отриманні токена html стан змінюється на "before html", після чого відбувається повторна обробка токена в цьому стані. В результаті створюється елемент HTMLHtmlElement, який додається до кореневого об'єкту Document. Стан змінюється на "before head". Аналізатор виявляє токен body. Хоча в нашому коді немає тега head, елемент HTMLHeadElement буде автоматично створений і доданий в дерево.

Стан змінюється на "in head", потім на "after head". Токен body обробляється ще раз, створюється елемент HTMLBodyElement, який додається в дерево, і стан змінюється на "in body".

Тепер прийшла черга токенів рядка Hello world. Виявлення першого з них веде до створення вузла Text, до якого потім додаються інші символи.

При отриманні останнього символу тесту, аналізатор закриває токена body і стан змінюється на "after body". Коли аналізатор доходить до закриваючого тега html, стан змінюється на "after after body". При отриманні токена кінця файлу аналіз завершується (рисунок 2.5).

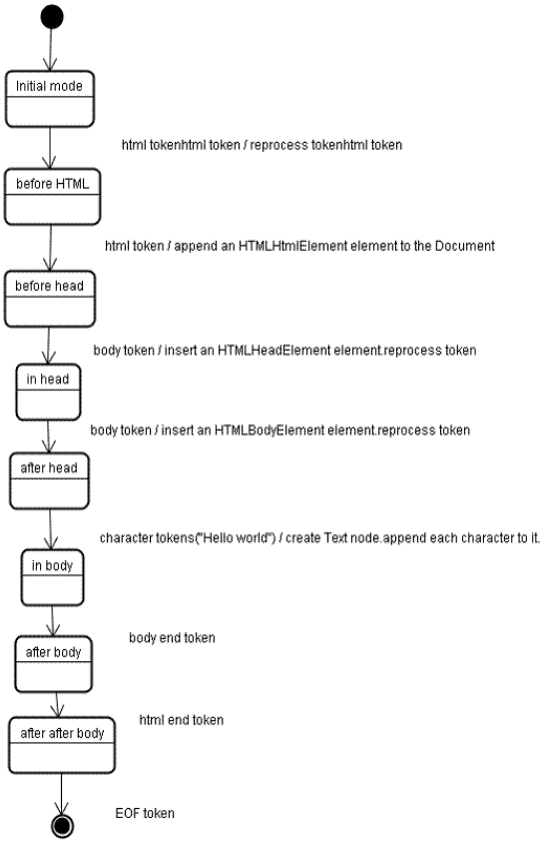


Рисунок 2.5 - Діаграма алгоритму побудови DOM дерева

2.2 Document API

Об'єкт Document, DOM моделі, надає доступ до широкого спектру методів для роботи з елементами DOM дерева. Як зазначено у роботах [1] - [2], це дозволяє змінювати структуру DOM під час виконання коду додатку і є основою розробки односторінкових додатків у браузері.

Як зазначено у роботі [4], виклики Document API, наприклад додавання нового елементу до сторінки або зчитування розмірів елементу, викликає повторне відображення у браузері, що потребує багато ресурсів, тому слід використовувати Document API лише у разі дійсної потреби.

Для того щоб керувати елементами всередині DOM, спочатку потрібно вибрати його і зберегти посилання на нього всередині змінної.

Наприклад, ви можемо вибрати елемент DOM дерева що є тегом a:

```
const link = document.querySelector ('a');
```

Тепер коли є посилання на елемент, що зберігається в змінній link, можна почати маніпулювати нею з використанням доступних властивостей і методів (які визначені в таких інтерфейсах, як HTMLAnchorElement в разі тегу a, його більш загальний батьківський інтерфейс HTMLElement та Node - який представляє всі вузли в DOM). Перш за все, змінимо текст всередині посилання, оновивши значення властивості Node.textContent. Потрібно додати наступний рядок нижче попередньої:

link.textContent = 'Mozilla Developer Network';

Також необхідно змінити URL-адресу, на який вказує посилання, щоб він не потрапляв в неправильне місце при натисканні:

link.href = “”;

Існує безліч способів вибору елемента і зберігання посилання на нього в змінної. Document.querySelector() - рекомендований сучасний метод, який вважається зручним, тому що він дозволяє вибирати елементи за допомогою селекторів CSS. Вищезгаданий виклик querySelector(‘a’) буде відповідати першому елементу <a>, який з'являється в документі. Якщо необхідно поєднати і робити щось з декількома елементами, можна використовувати Document.querySelectorAll(), який відповідає кожному елементу документа, який в свою чергу відповідає селектору і зберігає посилання на елемент в масиві масиво-подібному об'єкті, так званому NodeList.

Існують більш старі методи для захоплення посилань на елементи.

Document.getElementById(), який вибирає елемент із заданим значенням атрибута id: <p id = "myId"> Мій абзац </ p>

Ідентифікатор передається функції як параметр, тобто:

const elementRef = document.getElementById ('myId');

Document.getElementsByTagName(), який повертає масив, що містить всі елементи на сторінці даного типу, наприклад <p>, <a> і т.д. Тип елемента передається до функції в якості параметра, тобто:

const elementRefArray = document.getElementsByTagName ('p');

Створення і розміщення нових вузлів.

Захоплення посилання на елемент <section>:

```
const section = document.querySelector('section');
```

Створення нового абзацу, використовуючи Document.createElement():

```
var paragraph = document.createElement('p');
```

Додавання текстового вмісту:

```
paragraph.textContent = 'We hope you enjoyed the ride.';
```

Додавання новий абзац в кінці розділу, використовуючи Node.appendChild ():

```
section.appendChild(paragraph);
```

Створення текстового вузла, використовуючи Document createTextNode():

```
var textNode = document.createTextNode ( 'Lorem ipsum dolor sit amet, consectetur');
```

Використовуємо посилання на абзац, в якому знаходиться посилання, і додаємо до нього текстовий вузол:

```
var linkParagraph = document.querySelector ( 'p');
linkParagraph.appendChild (text);
```

Переміщення і видалення елементів.

Коли потрібно перемістити вузли або взагалі видалити їх з DOM ми можемо використати наступні методи. Для переміщення посиланням в середину абзацу “section” потрібно повторно викликати метод Node.appendChild():

```
section.appendChild(linkParagraph);
```

Для створення копії потрібно використовувати метод Node.cloneNode(). Для видалення вузла з батьківського елементу необхідно використовувати метод Node.removeChild() наприклад:

```
section.removeChild(linkParagraph);
```

Завдяки властивості parentNode видалення вузла може бути реалізовано за умови наявності посилання на дочірній елемент:

linkParagraph.parentNode.removeChild (linkParagraph);

Для загального маніпулювання HTML атрибутами є Element.setAttribute() що приймає два аргументи, назву атрибуту, який потрібно встановити для елемента, і значення, яке необхідно для установки нього:

paragraph.setAttribute ('class', 'highlight');

2.3 Віртуальна модель DOM

Віртуальна модель DOM (VDOM) - це модель, в якій віртуальне представлення інтерфейсу користувача зберігається в пам’яті і синхронізується з фактичним DOM в момент побудови або внесення змін до ного. Цей процес називається узгодженням.

Узгодження - це алгоритм, що лежить в основі віртуальної моделі DOM. Опис високого рівня алгоритму: коли браузер завантажує додаток, дерево вузлів, що описує додаток, генерується і зберігається в пам'яті. Потім це дерево передається в середовище візуалізації - наприклад, у випадку браузерного додатку переводиться в набір операцій DOM. Коли додаток оновлюється, генерується нове дерево. Нове дерево відрізняється від попереднього, тому щоб вичислити, які операції необхідні для оновлення інтерфейсу додатку, створюється масив змін. Далі, в момент коли потрібно відобразити зміну у фактичному DOM, залишається обійти масив змін та за допомогою викликів DOM API оновити фактичний DOM.

Розглянемо DOM дерево на рисунку 2.6.

Рисунок 2.6 - Дерево HTML

Це дерево також може бути представлено у вигляді об'єкта Javascript. В якому кожен объект буде мати наступні властивості:

- tagName - строкове значення, що буде відображати ім'я HTML тегу элементу DOM;
- children - масив для зберігання нащадків даного элемента DOM;
- attributes - об'єкт для зберігання HTML атрибутів DOM элемента, в якому ключем буде назва атрибуту, а значенням - значенням атрибуту;
- textContent - строкове значення, що буде відображати текст элемента DOM.

Представлення DOM дерева у вигляді об'єкта Javascript:

```
const vdom = {
    tagName: "html",
    children: [
        { tagName: "head" },
        {
            tagName: "body",
            children: [
            {
            tagName: "ul",
            attributes: { "class": "list" },
```

```
                children: [
                    {
                    tagName: "li",
                    attributes: { "class": "list__item" },
                    textContent: "List item"
                    }
                ]
                }
            ]
            }
        ]
};
```

Можемо вважати цей об’єкт і є наш віртуальний DOM. Як і оригінальний DOM, це об'єктне представлення нашого HTML-документа, але оскільки це звичайний об’єкт Javascript, ми можемо маніпулювати ним довільно та так часто як це буде потрібно, не торкаючись фактичного DOM, поки ми не захочемо їх синхронізувати, тобто внести зміни до фактичного DOM.

Замість того, щоб використовувати один об’єкт для всієї віртуальної моделі, краще працювати з невеликими частинами віртуального DOM, у вигляді різних JavaScript об'єктів. Таким чином зменшивши глибину вкладеності об'єктів ми зможемо добитися кращої продуктивності нашого коду, це значить що внесення змін до нашого віртуального DOM, та його синхронізація з фактичним DOM буде відбуватися швидше. Наприклад, ми можемо працювати над компонентом списку, що може бути представлений наступним JavaScript об'єктом:

```
const list = {
    tagName: "ul",
    attributes: { "class": "list" },
    children: [
```

```
        {
        tagName: "li",
        attributes: { "class": "list__item" },
        textContent: "List item"
        }
    ]
};
```

Тепер ми можемо використовувати віртуальний DOM для виділення конкретних змін, які необхідно внести в фактичний DOM, і зробити ці конкретні оновлення самостійно. Повернемося до нашого прикладу списку та внесемо в нього зміни. Наприклад змінимо текст першого елементу списку з “List item” на “List item one”, та додамо новий елемент списку що буде мати текст “List item two” та атрибут “class” зі значенням “list__item”.

Перше, що нам потрібно зробити, це скопіювати об'єкт віртуального DOM, що містить зміни, які ми хочемо внести. Оскільки нам не потрібно використовувати DOM API, ми можемо просто створити новий об’єкт, що буде містити старі змінені та нові елементи нашого DOM.

```
const copy = {
    tagName: "ul",
    attributes: { "class": "list" },
    children: [
    {
        tagName: "li",
        attributes: { "class": "list__item" },
        textContent: "List item one"
    },
    {
        tagName: "li",
        attributes: { "class": "list__item" },
```

```
            textContent: "List item two"
        }
        ]
};
```

Ця копія використовується для створення того, що називається "різницею" між фактичним та віртуальним DOM, у цьому випадку списком, та його оновленим представленням. Різниця може бути представлена у вигляді JavaScript масиву, де кожен елемент масиву будет об'єктом з полями:

а) newNode - нова версія DOM елемента;

б) oldNode - стара версія DOM елемента;

в) index - індекс елемента в масиві children нашого списку.

Приклад масиву різниці:

```
const diffs = [
        {
            // нова версія першого елемента списку
            newNode: {},
            // стара версія першого елемента списку
            oldNode: {},
            index: /* індекс елемента в масиві children списку */
        },
        {
            newNode: { /* новий елемент списку */ }
        }
];
```

Ця різниця надає інформацію щодо оновлення фактичного DOM. Після того, як всі різниці зібрані, ми можемо оновити фактичний DOM, оновивши при цьому лише необхідні його елементи.

Наприклад, ми можемо циклічно обійти усі масиви різниць та додавати новий, видаляти, або оновлювати старий елемент, залежно від того, що вказано в різниці.

Приклад оновлення списку:

```
// поруш елементу списка в фактичному DOM
const domElement = document.getElementsByClassName("list")[0];

// обхід масиву змін
diffs.forEach(diff => {
    // створення нового елемента
    const newElement = document.createElement(diff.newNode.tagName);
    if (diff.oldNode) {
        // заміна елементу
        domElement.replaceChild(diff.newNode, diff.index);
    } else {
        // додавання нового елементу списка
        domElement.appendChild(diff.newNode);
    }
});
```

Діаграма класів реалізації VDOM зображена на рисунку 2.8.

Також віртуальну модель DOM можна представити у вигляді графу (рисунок 2.7), де вершинами графу будуть елементи моделі, а ребра графу будуть відображають зв'язки між дочірнім та батьківським елементом.

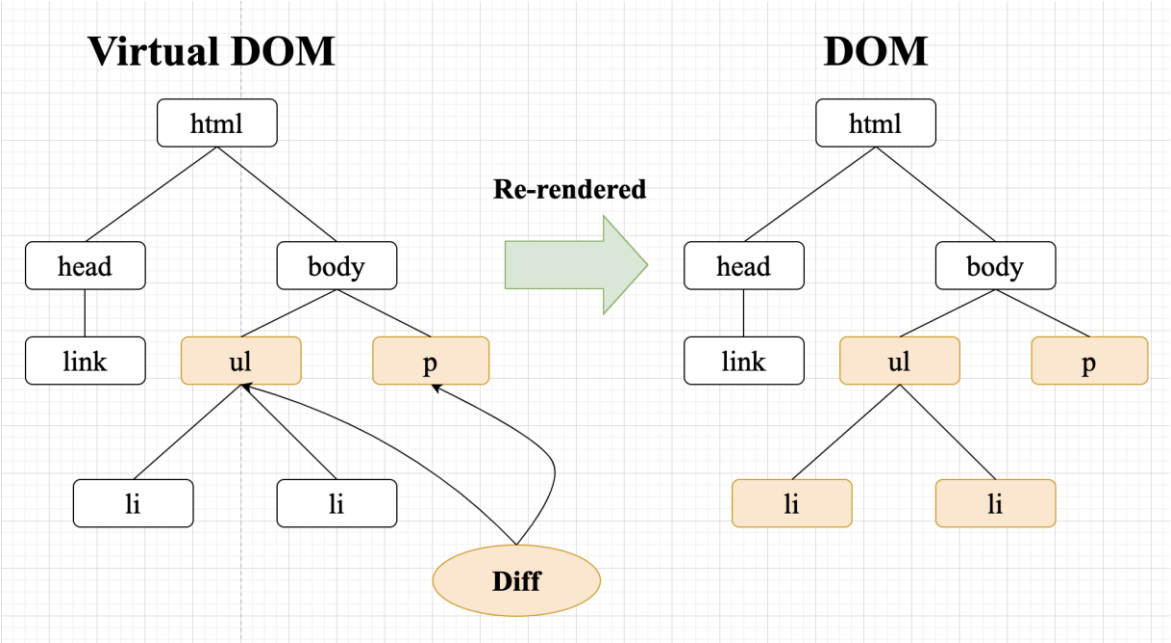


Рисунок 2.7 - Репрезентація віртуальної та фактичної DOM моделей у вигляді графів

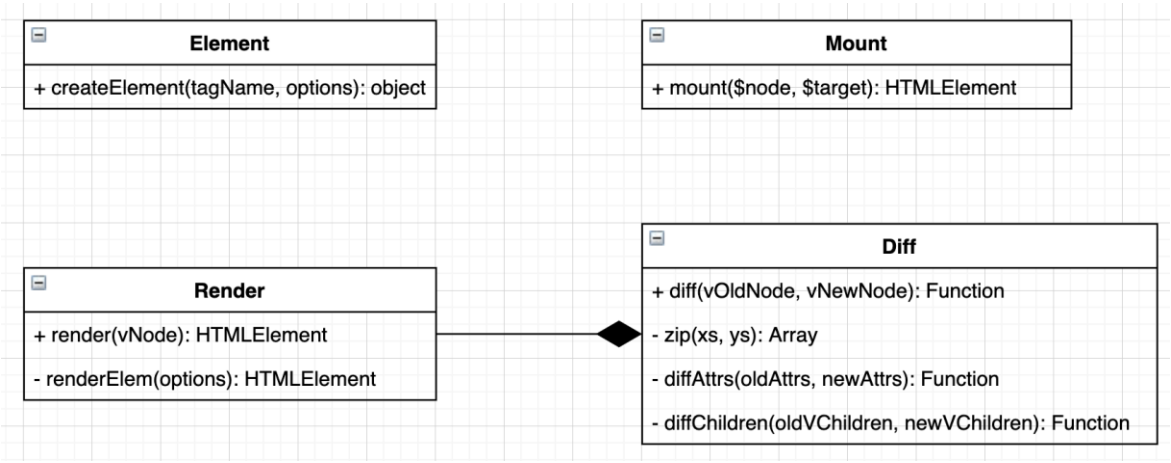


Рисунок 2.8 - Діаграма класів реалізації VDOM

Як висновок віртуальна модель DOM зберігає фактичну модель DOM та зміни які повинні відбутися в ньому в пам'яті, це дає можливість контролювати процес оновлення фактичної DOM, швидко оновлювати тільки певні елементи, оновлення за пріоритетом. Також вона дозволяє абстрагувати усі операції з властивостями, обробку явищ та внесення змін до DOM вручну, які інакше доведеться реалізовувати для створення web-додатків. З іншого боку такий підхід потребує великої кількості память, тому що в реальних проектах DOM дерево може містити тисячі елементів.

2.4 Інкрементальна модель DOM

Інкрементальна модель DOM намагається запропонувати більш простий підхід до побудови та зміни DOM ніж альтернативи. Замість того, щоб зберігати повну копію фактичного DOM як JavaScript об'єкта та зберігати масиви змін, в інкрементальній моделі внемення змін елементи фактичного DOM відбувається на пряму за допомогою Document API.

Слід зазначити що у більш сучасних реалізація на етапі компіляції додатку, кожен компонент компілюється в набір інструкцій, що описують створення елементів та внесення зміни тільки до певних атрибутів та властивостей елементів, які передбачені логікою компоненти. В ранніх реалізаціях інкрементальної моделі створення інструкцій на етапі компіляції було не можливо, тому до кожної компоненти додавався стандартний набір інструкцій для зміни елемету DOM дерева в залежності від шаблону елемента.

Інкрементальна модель DOM, не використовує додаткову, віртуальну, копію DOM, що призводить до зменшення використання пам'яті. Але в свою чергу це призводить до зниження швидкості пошуку відмінностей тому що набір інструкцій компоненти складається з викликів Document API. Слід зазначити, що додавання або видалення елементів з фактичного DOM відбувається швидше ніж у віртуальній моделі.

Зменшене використання пам’яті є ключовим фактором для мобільних пристроїв або інших пристроїв з обмеженою пам’яттю.

Розглянемо приклад DOM дерева рисунку 2.6.

Наприклад ми хочемо внести зміни до списку, змінити текст першого елементу списку з “List item” на “List item one”, та додати новий елемент списку що буде мати текст “List item two” та атрибут “class” зі значенням “list__item”.

Припустимо що на етапі розробки ми створили компонент List що будує відображення списку елементів з масиву items.

```
// масив елементів
```

```
const items = ['List item'];
// компонент List
const List = elementOpen('ul', null, {class: 'list'});
        items.forEach(item => {
                elementOpen('li', item, {class: 'list_item'});
                text(item);
                elementClose('li');
        });
elementClose('ul');
```

При кожному виклику функції elementOpen відбувається перевірка фактичного DOM на наявність даного нащадка в батьківському елементі. У випадку якщо це елемент був створений раніше, відбувається його оновлення, у випадку якщо такого елемента не існує в фактичному DOM, буде створено новий елемент, який буде додано до батьківського елемента.

Далі ми копіюємо та додаємо новий елемент до масиву items.

```
const newItems = [...items];
newItems.push('List item two');
```

Для внесення змін до фактичного DOM ми повинні викликати функцію patch, що приймає посилання на елемент списку та функцію оновлення.

```
patch(List, () => {
        newItems.forEach(item => {
                elementOpen('li', item);
                text(item);
                elementClose('li');
        });
});
```

Функція patch є обгорткою що визначає батьківський елемент, викликає функцію оновлення та видаляє старі елементи з фактичного DOM.

```
const patch = ($parent, patcher) => {
```

```
        currentParent = $parent;

        if (currentParent._idom) {
            currentParent._idom.childrenKeys = [];
        }

        patcher();

        removeExcessChildren(currentParent);
};
```

Як висновок інкрементальна модель DOM змушує викликати DOM API для перевірки стану фактичного DOM. В даному прикладі усі виклики DOM API знаходяться в одному синхронному стеці, тому будуть виконані по черзі і лише коли синхронний стек буде очищено браузер відтворить результат на екрані. Але в реальних додатках, за такої моделі, неможливо згрупувати усі виклики в одих стек, а навіть якщо і можливо, то час виконання задач цього стеку при кожній зміні моделі DOM буде значно більшим за час відтворення одного кадру браузером. Результатом цього може блокування синхронного стеку виконання JavaScript коду, що в свою чергу призведе до порушення процесу відмальовування, зменшення частоти кадрів відображення та появи візуальних дефектів в інтерфейсі додатку.

Діаграма класів реалізації IDOM зображена на рисунку 2.11.

Деякі бібліотеки що реалізую дану модель оптимізують виклики DOM API, складаючи їх в окремий стек, та виконуючи ці виклики групами, це частково вирішує проблему блокування синхронного JavaScript стеку, але не повністю.

Також інкрементальна модель DOM можна представити у вигляді графу (рисунок 2.10), де вершинами графу будуть елементи моделі, а ребра графу будуть відображають зв'язки між дочірнім та батьківським елементом.

Рисунок 2.10 - Репрезентація віртуальної та фактичної DOM моделей у вигляді графів

Рисунок 2.11 - Діаграма класів реалізації IDOM

Висновок до розділу

В даному розділі було розглянуто DOM, наведено приклади методів для побудови та зміни DOM за допомогою Document API. Розглянуто IDOM та VDOM моделі. Представлена діаграма класів власних реалізацій IDOM та VDOM.

3 ПОБУДОВА DOM ЗА ДОПОМОГОЮ JAVASCRIPT БІБЛІОТЕК

3.1 Аналітичний огляд бібліотеки React

React - це JavaScript-бібліотека для створення користувацьких інтерфейсів що використовує віртуальну модель DOM.

Вона дозволяє в декларативному стилі робити виклики API бібліотеки за допомогою яких можна повідомити React, в якому стані розробник хоче, щоб інтерфейс користувача знаходився, і він гарантує, що DOM буде відповідати цьому стану.

React реалізує віртуальну модель DOM та доповнює її певними особливостями.

3.1.1 Архітектура Fiber та реалізація віртуальної моделі DOM в React

У роботах [10] та [12] зазначено, що React Fiber - це реалізація основного алгоритму роботи бібліотеки React.

Мета React Fiber - збільшити її придатність для таких областей, як анімація та жести. Її головною особливістю є поступове відображення - можливість розділити відображення на шматки та розподілити її відмальовування на кілька кадрів. Інші ключові особливості включають:

- можливість призупинення, переривання або повторного використання результатів обчислень, у разі надходження нового оновлення DOM;
- можливість призначати пріоритет різним типам оновлень.

Основна ідея API React - реалізувати алгоритм синхронізації віртуальної та фактичної моделі DOM так, ніби вона спричиняє повторне відмальовування всього додатка. Як зазначено у роботі [13], це дозволяє розробнику міркувати декларативно, а не турбуватися про те, як ефективно перевести додаток із будь-якого конкретного стану в інший (від A до B, B до C, C до A тощо).

Насправді повторне відображення всього додатка при кожній зміні працює лише для самих тривіальних додатків, у реальному додатку це надзвичайно дорого з точки зору продуктивності. React реалізує оптимізації, які створюють вигляд зміни відображення всього додатка, зберігаючи при цьому високу продуктивність. Основна частина цих оптимізацій є частиною процесу узгодження віртуальної моделі DOM.

Якщо якісь елементи інтерфейсу є поза екраном, React може затримати будь-яку логіку, пов’язану з цими елементами. Якщо дані надходять швидше, ніж частота кадрів, React може об'єднати та обробити зміни як пакетні оновлення. React може визначити пріоритет оновленням, якщо це випливає на взаємодію користувача з інтерфейсом додатку, щоб уникнути зменшення кількості кадрів. Наприклад, заблокувати оновлення анімації, спричиненої натисканням кнопки, на користь більш важливому оновлення, наприклад, надання нового вмісту, щойно завантаженого з мережі.

Також слід зазначити, що DOM - це лише одне з середовищ візуалізації, з яким може працювати React, інші - це платформи iOS та Android, за допомогою бібліотеки React Native.

Причина, по якій він може підтримувати так багато середовищ, полягає в тому, що React створений таким чином, що узгодження та відтворення є окремими фазами. У фазі узгодження React виконує роботу з обчислення того, які частини дерева змінилися, а потім у фазі відтворення використовує цю інформацію для фактичного оновлення наданого додатку.

Цей поділ означає, що бібліотеки React DOM та React Native можуть використовувати власні візуалізатори під час спільного використання одного і того ж узгоджувача, наданого ядром React.

З точки зору розробника кожен Fiber вузол імплементує інтерфейс об'єкта React.Component, що має метод “setState”, викликавши цей метод розробник може запустити процес оновлення стану даного Fiber вузла, що в свою чергу призведе до оновлення моделі DOM та оновлення відображення.

У своїй поточній реалізації React ходить по дереву віртуальної моделі DOM рекурсивно і викликає функції візуалізації всього оновленого дерева під час одного кадру.

Існує 4 основні функції, які використовуються для переміщення по React дереву, ініціювання або завершення обчислень та процесу узгодження:

1) performUnitOfWork;
2) beginWork;
3) completeUnitOfWork;
4) completeWork.

Кожна функція приймає Fiber вузол для обробки, і в міру того, як React спускається вниз по дереву, ви можете побачити зміни активного в даний час Fiber вузла. На рисунку 3.1, ви можете чітко бачити, як алгоритм переходить від однієї гілки до іншої. a1, b1, b2, b3, c1, c2, d1 та d2 є компонентами інтерфейсу. Спочатку вона завершує обчислення для дітей, а потім переходить до батьків.

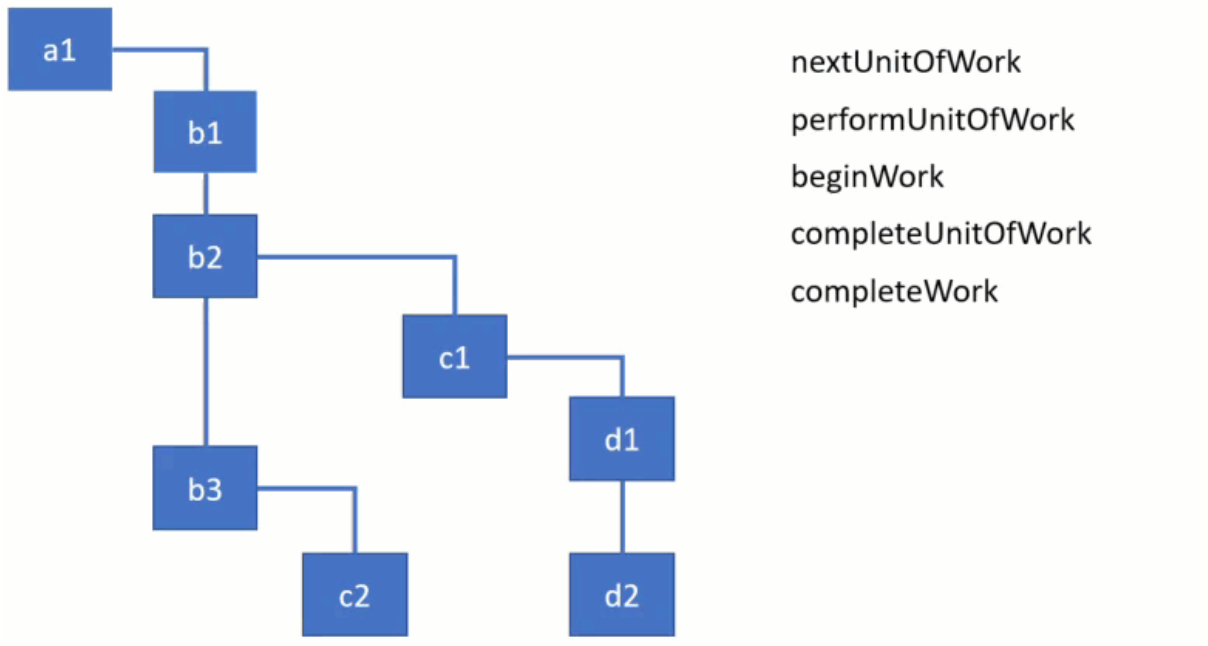


Рисунок 3.1 - Обхід дерева Fiber вузлів

Функція performUnitOfWork, отримує Fiber вузол з дерева workInProgress і запускає обчислення, викликаючи функцію beginWork. Як зазначено у роботі [17]:

```
function performUnitOfWork(workInProgress) {
    let next = beginWork(workInProgress);
    if (next === null) {
        next = completeUnitOfWork(workInProgress);
    }
    return next;
}

function beginWork(workInProgress) {
    console.log('work performed for ' + workInProgress.name);
    return workInProgress.child;
}
```

performUnitOfWork функція яка запустить всі дії, які необхідно виконати для Fiber вузла. Для цілей цієї демонстрації ми просто записуємо назву вузла, щоб вказати, що обчислення були виконані. Функція beginWork завжди повертає покажчик на наступну Fiber вузол або null. Якщо є наступний дочірній елемент, то він буде присвоєний змінній nextUnitOfWork в функції workLoop. Однак, якщо дочірнього елемента немає, згідно алгоритму, він досяг кінця гілки і може завершити обчислення над поточним елементом. Як зазначено у роботі [18]:

```
function workLoop(isYieldy) {
    if (!isYieldy) {
        while (nextUnitOfWork !== null) {
        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
    }
}
```

Після того, як обчислення буде завершено, йому потрібно буде виконати обчислення для сусідніх елементів і повернутися до батьків. Це відбувається в функції completeUnitOfWork. Важливо розуміти, що на даний момент React

завершив обчислення тільки для попередніх сусідніх елементів, але не завершив обчислення для батьківського елемента. Як зазначено у роботі [18]:

```
function completeUnitOfWork(workInProgress) {
        while (true) {
        let returnFiber = workInProgress.return;
        let siblingFiber = workInProgress.sibling;

        nextUnitOfWork = completeWork(workInProgress);

        if (siblingFiber !== null) {
                // If there is a sibling, return it
                // to perform work for this sibling
                return siblingFiber;
        } else if (returnFiber !== null) {
                // If there's no more work in this returnFiber,
                // continue the loop to complete the parent.
                workInProgress = returnFiber;
                continue;
        } else {
                // We've reached the root.
                return null;
        }
        }
}

function completeWork(workInProgress) {
        console.log('work completed for ' + workInProgress.name);
        return null;
}
```

Тільки після того, як обчислення над усіма гілками, починаючи з дочірніх, буде завершено, він завершить обчислення для батьківського елемента.

Як видно з реалізації, performUnitOfWork і completeUnitOfWork, використовуються в основному для ітерацій, в той час як основні дії виконуються у функціях beginWork і completeWork.

Рисунок 3.2 - Запис процесу оновлення DOM та відображення за допомогою React та Fiber архітектури

Виходячи з рисунку 3.2 можна побачити що всі оновлення відбуваються поступово та головний потік JavaScript не є заблокованим більше ніж на час відмальовування одного кадру браузером.

3.2 Аналітичний огляд фреймворку Angular

Angular - це фреймворк для побудови односторінкових клієнтських додатків за допомогою HTML та JavaScript. Як зазначено у роботі [14], він реалізує шаблон проектування MVVM (Model-View-View-Controller).

Архітектура додатків побудованих на базі Angular спирається на певні фундаментальні концепції. Основними будівельними блоками є екземпляри

класу NgModule, які забезпечують контекст компіляції компонентів. У додатку завжди є принаймні кореневий модуль, який дозволяє завантажувати функціональні модулі.

Модуль NgModule складається з наступних елементів:

- шаблон (Template) - виглядає як звичайний HTML, за винятком того, що він містить синтаксис Angular шаблону. Шаблон може використовувати функцію прив'язки даних для узгодження даних додатку та DOM;
- вид (View) - описує певну частину інтерфейса додатку та складається з Template;
- компонент (Component) - відповідає за контроль певної частини інтерфейсу що складається з View;
- сервіс (Service) - це широка категорія, що включає будь-яке значення, функцію чи набір функцій, які потрібні додатку. Service, як правило, є класом з вузькою, чітко визначеною метою.

Оскільки метою даної дипломної роботи є аналіз моделей побудови DOM, далі будуть розглянуті тільки Template, View та Component елементи бібліотеки Angular які працюють безпосередньо з DOM.

3.2.1 Реалізація інкрементальної моделі DOM в Angular

Як зазначено у роботі [15], Template та View бібліотеки Angular реалізують інкрементальну модель побудови та зміни DOM. Побудуємо компонент списку:

```
Component({
    selector: '.list_item',
    template: `
        <li>
            {{text}}
        </li>`
})
```

```
class ListItemComponent {
    text: Observable<string> = this.store.pipe(select('text'));
    constructor(private store: Store<AppState>) {}
}
```

Цей компонент буде скомпільовано у наступний код:

```
var ListItemComponent = /** @class */ (function () {
    function ListItemComponent(store) {
        this.store = store;
        this.text = this.store.pipe(select('text'));
    }

    ListItemComponent.componentDef = defineComponent({
        type: ListItemComponent,
        selectors: [[".list_item"]],
        factory: function ListItemComponent_Factory(t) {
        return new (t || ListItemComponent)(directiveInject(Store));
    },
    consts: 2,
    vars: 3,
    template: function ListItemComponent_Template(rf, ctx) {
    // створення DOM
    if (rf & 1) {
        pipe(1, "async");
        template(0, ListItemComponent_li_Template_0, 2, 1, null, _c0);
    // зміна DOM
    } if (rf & 2) {
        elementProperty(0, "textContent", bind(pipeBind1(1, 1, ctx.todos)));
    }
    },
```

```
    encapsulation: 2
    });

return ListItemComponent;
}());
```

Як ми можемо побачити, скомпільований код описує зміну тексту елементу списку, адже це єдине що буде змінено в цьому елементі DOM.

У випадку якщо нам потрібно додати або видалити елемент DOM ми можемо зробити це напряму без додаткових перевірок, але у випадку зміни тексту чи атрибутів елементу, спочатку нам потрібно перевірити чи потрібно нам оновлювати фактичну DOM модель. а вже потім вносити в нього потрібні зміни. Функція template приймає аргумент rf на основі якого перевіряється який тип операції, побудова або зміна DOM, необхідно виконати. При операції зміни буде викликано функцію elementProperty яка перевірить чи потрібно вносити зміни у фактичний DOM перевіривши текст елементу за допомогою DOM API.

Angular використовує прив’язки(bindings) для визначення залежностей кожного вузла DOM від властивостей класу компонентів. Під час виявлення змін кожна прив'язка визначає тип операції, яку Angular повинен використовувати для оновлення вузла. Тип операції визначається прапорами прив'язки(binding flags). Наприклад, для типових операцій DOM він складається зі списку що наведений у таблиці 3.1.

Таблиця 3.1 - Типові DOM операції

Тип операції	Конструкція в шаблоні
TypeElementAttribute	attr.name
TypeElementClass	class.name
TypeElementStyle	style.name

Функція updateRenderer приймає два параметри _ck і v. Параметр _ck посилається на функцію prodCheckAndUpdate. Параметр v посилається на View компонента з вузлами. Функція updateRenderer виконується кожного разу, коли Angular запускає перевірку виявлення змін для компонента. Як зазначено у роботі [19]:

```
function updateRenderer(_ck, _v) {
    var _co = _v.component;
    var currVal_0 = _co.name;
    _ck (_v, 1,0, currVal_0);
```

Основне завдання функції updateRenderer - отримати поточне значення привязаної властивості з экземпляра компонента та викликати функцію _ck, передав їй View, індекс вузла та отримане значення. Важливо зазначити, що Angular виконує оновлення DOM для кожного вузла окремо - тому використовується індекс вузла. Ви можете це чітко побачити під час перегляду списку параметрів функції prodCheckAndUpdateNode, на яку посилається _ck в залежності від типу вузла DOM [20]:

```
function prodCheckAndUpdateNode (
    view: ViewData,
    nodeIndex: number,
    argStyle: ArgumentType,
    v0 ?: any,
    v1 ?: any,
    v2 ?: any,
```

nodeIndex - це індекс вузла view, для якого слід виконати виявлення змін.

Якщо у шаблоні додатку є кілька виразів:

```
<h1> Привіт {{name}} </h1>
<h1> Привіт {{age}} </h1>
```

Компілятор буде генерувати наступне тіло функції updateRenderer для кожного з вузлів h1:

```
var _co = _v.component;
// тут індекс вузла дорівнює 1, а властивість - “name”
var currVal_0 = _co.name;
_ck (_v, 1,0, currVal_0);
// тут індекс вузла дорівнює 4, а привязана властивість - “age”
var currVal_1 = _co.age;
_ck (_v, 4,0, currVal_1);
```

У функцію updateRenderer передається _ck. Під час виявлення змін - цей параметр посилається на фунцкію prodCheckAndUpdate. Це коротка загальна функція, яка здійснює купу викликів в залежності від типу DOM вузла, які у кінці алгоритму викликають функцію checkAndUpdateNode.

Функція checkAndUpdateNode - це маршрутизатор, який розрізняє типи вузлів DOM а делегати перевіряють та оновлюють дані цих вузли відповідно до даних компонент. Частінита коду функції:

```
case NodeFlags.TypeElement -> checkAndUpdateElementInline
case NodeFlags. TypeText -> checkAndUpdateTextInline
case NodeFlags.TypeDirective -> checkAndUpdateDirectiveInline
```

Як ми бачимо ElementInline, TextInline та DirectiveInline це різні типи компонент.

В свою чергу функції checkAndUpdateElementInline, checkAndUpdateTextInline та checkAndUpdateDirectiveInline виконують перевірку відповідності даних компоненти і вузла реальної моделі DOM, та змінюють його у випадку невідповідності.

Висновок до розділу

В даному розділі було розглянуто високорівневу архітектуру та моделі побудови і зміни DOM що імплементують JavaScript бібліотеки React та Angular.

4 КОМПЛЕКС ЗАДАЧ АНАЛІЗУ МОДЕЛЕЙ ПОБУДОВИ DOM

Комплекс задач складається із задач аналізу швидкості побудови та зміни DOM та задач оцінки використаної пам'яті.

Задачі аналізу швидкості побудови та зміни DOM:

1) Створення таблиці що містить 1000 рядків;
2) Створення таблиці що містить 10000 рядків;
3) Оновлення таблиці що містить 1000 рядків;
4) Видалення таблиці що містить 10000 рядків;
5) Оновлення тексту кожного 10-го рядка для таблиці що містить 10000 рядків;
6) Видалення одного рядка з таблиці що містить 1000 рядків у відповідь на натискання на рядок;
7) Заміна місцями 2 рядків в таблиці що містить 1000 рядків;
8) Видалення одного рядка з таблиці що містить 1000 рядків;
9) Додавання 1000 рядків до таблиці що містить 10000 рядків.

Задачі оцінки використаної пам'яті:

1) Створення таблиці що містить 1000 рядків;
2) Оновлення кожного 10 рядка таблиці що містить 1000 рядків;
3) Додавання 1000 рядків до існуючої таблиці;
4) Створення та видалення таблиці що містить 1000 рядків.

Для вирішення кожної із задач було створено страндарну HTML сторінку, що має один елемент div, який буде контейнером нашого додатку, тег script за допомогою якого підключається файл з JavaScript кодом.

Розглянемо приклад HTML сторінки, що згенерований середовищем розробки WebStorm та зазначений у роботі [21]:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport"
    content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <div id="root"></div>
    <script src="./dist/main.js"></script>
    <script src="./task.js"></script>
</body>
</html>
```

У файлі main.js знаходиться скомпільований код власної реалізації IDOM або VDOM.

У файлі task.js ми описуємо інструкції побудови DOM.

4.1 Задачі аналізу швидкості побудови та зміни DOM

Вирішення задачі №1, побудова таблиці що містить 1000 рядків:

```
// імпортуємо методи власної реалізації IDOM
const {elementOpen, elementClose, text, patch} = idom;
// зберігаємо посилання на контейнер нашого додатку
const root = document.getElementById('root');
// генеруємо двовимірний масив що містить 1000 масивів, які будуть рядками
нашої таблиці, кожен з них містить 10 елементів - комірки таблиці
const tableItems = new Array(1000).fill(undefined).map((a, i) => {
   return new Array(10).fill(undefined).map((b, j) => 'item ' + i + ' ' + j);
});


// створюємо компонент TR
```

```
// функція що приймає масив комірок одного рядка таблиці
const TR = (tds) => {
	tds.forEach(td => {
		elementOpen('td', td);
			text(td);
		elementClose('td');
	});
};

// створюємо компонент Table
// функція що приймає масив рядків таблиці
const Table = (trs) => {
	elementOpen('table');
	trs.forEach(tr => {
		elementOpen('tr', tr.join(' '));
			TR(tr);
		elementClose('tr');
	});
	elementClose('table');
};

// викликати функцію patch, що побудує фактичний DOM
patch(root, () => {
	Table(tableItems);
});
```

Результатом виконання даного JavaScript коду буде HTML сторінка яка містить таблицю, що зображена на наступному рисунку 4.1.

Рисунок 4.1 - Сторінка після побудови таблиці

Як зазначено у роботі [16], за допомогою Chrome DevTools, у вкладці Performance, ми можемо зробити Snapshot (рисунок 4.2) що буде містити інформацію про процес виконання JavaScript коду та побудову HTML від початку завантаження сторінки до довільного моменту у часі. За допомогою Snapshot ми вимірюємо час за який таблиця була побудована кожною з реалізацій.

Рисунок 4.2 - Приклад Snapshot у Chrome DevTools

Результати наведені у таблиці 4.1.

Таблиця 4.1 - Результати виконання задачі №1

Задача	IDOM Angular (МС)	VDOM React (МС)	IDOM (Власна реалізація) (МС)	VDOM (Власна реалізація) (МС)
Тривалість створення таблиці що містить 1000 рядків	185.21	180.57	350.3	278.6

Вирішення задачі №3, оновлення таблиці що містить 1000 рядків:

Для оновлення таблиці ми можемо використати код вирішення задачі №1 але додати до нього таймер що викличе оновлення таблиці через 2 секунди після побудови.

```
setTimeout(() => {
    // створюємо новий двовимірний масив таблиці
    const newItems = new Array(1000).fill(undefined).map((a, i) => {
        return new Array(10).fill(undefined).map((b, j) => 'new item ' + i + ' '
    + j);
    });
    // оновлюємо таблицю за посиланням на елемент
    patch(tableElement, () => {
        newItems.forEach(tr => {
            elementOpen('tr', tr.join(' '));
                TR(tr);
            elementClose('tr');
        });
    });
}, 2000);
```

Спочатку відбудеться побудова початкового DOM (рисунок 4.3) і через 2 секунди кожен DOM елемент комірки таблиці буде оновлено (рисунок 4.4).

Рисунок 4.3 - Сторінка до оновлення таблиці

Рисунок 4.4 - Сторінка після оновлення таблиці

Результати вимірювань часу зображено у наступній таблиці 4.2.

Таблиця 4.2 - Результати виконання задачі №3

Задача	IDOM Angular (MC)	VDOM React (MC)	IDOM (Власна реалізація) (MC)	VDOM (Власна реалізація) (MC)
Тривалість оновлення всіх 1000 рядків таблиці	161.22	157.32	275.23	177.4

На прикладі вирішення задач №1 та №3 будуємо реалізації для виконання заданого комплексу задач для аналізу часу, результати наведені в таблиці 4.3.

Таблиця 4.3 - Результати виконання комплексу задач аналізу часу побудови і зміни DOM

Задача	IDOM Angular (MC)	VDOM React (MC)	IDOM (Власна реалізація) (MC)	VDOM (Власна реалізація) (MC)
1	2	3	4	5
Тривалість створення таблиці що містить 1000 рядків	185.21	180.57	350.3	278.6
Тривалість оновлення всіх 1000 рядків таблиці	161.22	157.32	275.23	177.4
Тривалість оновлення тексту кожного 10-го рядка для таблиці з 10000 рядків	68.83	81.92	101.5	143.5

Продовження таблиці 4.3

1	2	3	4	5
Тривалість видалення рядка у відповідь на натискання на рядок	7.94	10.32	18.4	25.6
Тривалість заміни місцями 2 рядків в таблиці на 1000 рядків	105.81	106.51	154.4	164.6
Тривалість видалення рядка	47.13	49.6	56.8	73.4
Тривалість створення таблиці що містить 10000 рядків	1693.3	1935.6	2304.6	2504
Тривалість додавання 1000 рядків до таблиці на 10000 рядків	243.36	268.66	405.4	479.1
Тривалість очищення таблиці, заповненої на 10000 рядків	263.93	175.44	432.4	269.6

Як висновок, ми можемо бачити що VDOM швидкий у задачах створення або видалення великої групи елементів, але у задачах зміни певних елементів у вже існуючій сторінці цей метод є повільнішим за IDOM.

4.2 Задачі аналізу використаної пам'яті при побудові DOM

На прикладі вирішення задач №1 та №3 аналізу часу побудови та зміни DOM будуємо реалізації для виконання задач аналізу кількості використаної пам'яті. Як і у попередніх задачах Snapshot з Chome DevTools дає змогу

відстежити кількість пам'яті що була використана. Результати вирішення задач оцінки використаної пам'яті наведені в таблиці 4.4.

Таблиця 4.4 - Результати виконання комплексу задач аналізу кількості використаної пам'яті при побудові чи зміні DOM

Задача	VDOM React (МБ)	IDOM Angular (МБ)	VDOM (Власна реалізація) (МБ)	IDOM (Власна реалізація) (МБ)
Кількість використаної пам'яті після створення таблиці що містить 1000 рядків	6.7	9.8	27.4	26.3
Кількість використаної пам'яті після оновлення кожного 10 рядка таблиці що містить 1000 рядків	7.6	9.8	9.8	5.3
Кількість використаної пам'яті після додавання 1000 рядків до існуючої таблиці	7.9	10.1	10.5	6.6
Кількість використаної пам'яті після створення та видалення таблиці що містить 1000 рядків	3.8	6.4	16.42	11.5

Як висновок, ми можемо бачити що VDOM React використовує менше пам'яті за IDOM Angular, але цей результат не є репрезентативним для нас, тому що, React це бібліотека яка займається лише роботою з DOM, а Angular

це фреймворк для побудови додатків що містить у собі велику кількість функції крім роботи з DOM. Більш репрезентативними є результати власних реалізацій, на основі цих даних ми можемо зробити висновок що VDOM використовую значно більше пам'яті ніж IDOM.

Висновок до розділу

На основі даних обох таблиць аналізу часу побудови та кількості використаної пам'яті ми можемо виділити основні факти:

VDOM:

- використовує більше пам'яті;
- зберігає стан DOM в пам'яті, що дає змогу швидко відлагоджувати роботу додатку;
- процес створення та видалення великої кількості зв'язаних елементів є більш швидким за IDOM.

IDOM:

- використовує менше пам'яті;
- має простіший API та є більш гнучким у інтеграції з існуючими додатками;
- процес оновлення елементів є більш швидким за VDOM.

ЗАГАЛЬНІ ВИСНОВКИ

У першому розділі було наведено опис роботи модуля відображення браузера та процесу побудови DOM.

У другому розділі було описано DOM та як за допомогою Document API можливо його змінювати. Також було описано існуючі моделі та методи побудови і зміни DOM, створено власні реалізації цих моделей.

У третьому розділі було наведено опис існуючих рішень, було обрано та досліджено їх реалізації моделей побудови DOM.

Четвертий розділ присвячений вирішенню заданого комплексу задач за допомогою власних реалізацій та існуючих JavaScript бібліотек. На основі зібраних даних було проведено порівняльний аналіз, що дає змогу вибрати оптимальних метод.

На основі результатів стає очевидно що віртуальна та інкрементальна моделі мають суттєві відмінності, тому може бути обрана як одна так і інша модель, в залежності від складності додатку та вимог до його використання.

На даному етапі застосування власних реалізацій, не може дати найкращий результат у порівнянні з існуючими рішеннями через відсутність певних оптимізацій у роботі алгоритмів або певних функціональних особливостей.

В майбутньому планується розширення кожної з реалізацій. Додавання можливості відтворення змін DOM частково, побудови початкової HTML сторінки на сервері, інтеграція з існуючими шаблонізаторами.

ПЕРЕЛІК ПОСИЛАНЬ

1. Resig J. Secrets of JavaScript Ninja / J. Resig, B. Bibeault., 2013. – 382 с.
2. Souders S. High Performance Web Sites: Essential Knowledge For Front-End Engineers / Steve Souders., 2007. – 168 с.
3. What is V8? [Електронний ресурс] – Режим доступу до ресурсу: https://v8.dev/
4. Performance Matters [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/web/fundamentals/performance
5. Render Performance [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/web/fundamentals/performance/rendering
6. Browser Engines [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@jonbiro/browser-engines-chromium-v8-blink-gecko-webkit-98d6b0490968
7. How browsers work [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work
8. Chromium [Електронний ресурс] – Режим доступу до ресурсу: https://chromium.googlesource.com/chromium/src/
9. DOM [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/
10. React [Електронний ресурс] – Режим доступу до ресурсу: https://uk.reactjs.org/
11. How Browsers Work: Behind the scenes of modern web browsers [Електронний ресурс] – Режим доступу до ресурсу: https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/
12. React Fiber Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/acdlite/react-fiber-architecture

13. Inside Fiber [Електронний ресурс] – Режим доступу до ресурсу: https://indepth.dev/inside-fiber-in-depth-overview-of-the-new-reconciliation-algorithm-in-react/
14. Angular [Електронний ресурс] – Режим доступу до ресурсу: https://angular.io/
15. Understanding Angular Ivy: Incremental DOM and Virtual DOM [Електронний ресурс] – Режим доступу до ресурсу: https://blog.nrwl.io/understanding-angular-ivy-incremental-dom-and-virtual-dom-243be844bf36
16. Debugging JavaScript in Chrome DevTools [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/web/tools/chrome-devtools/javascript
17. MayReactV16 [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/sven36/MayReactV16/blob/f0f9059b9851898efdcb56a75bf6f168509705a9/README.md
18. Funfish blog [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/funfish/blog/blob/master/28.%20react%20%E6%BA%90%E7%A0%81%E5%BC%80%E5%A7%8B%E7%9A%84%E9%82%A3%E4%B8%80%E6%AD%A5.md
19. One-way property binding mechanism in Angular [Електронний ресурс] – Режим доступу до ресурсу: https://blog.bitsrc.io/one-way-property-binding-mechanism-in-angular-f1b25cf00de7
20. The mechanics of DOM updates in Angular [Електронний ресурс] – Режим доступу до ресурсу: https://indepth.dev/the-mechanics-of-dom-updates-in-angular/
21. HTML [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/help/webstorm/editing-html-files.html

Додаток А

Тексти програмного коду

Реалізація моделей IDOM та VDOM

(Найменування програми (документа))

DVD-R

(Вид носія даних)

8 арк, 10 Кб

(Обсяг програми (документа) , арк.,) Кб)

Київ – 2020 року

attrs.js

```
const setAttrs = ($node, attrName, attrValue) => {
  if ($node.getAttribute(attrName) !== attrValue) {
    $node.setAttribute(attrName, attrValue);
  }
};

const eventListener = ($node, eventName, eventListener) => {
  if ($node._idom && $node._idom.events[eventName]) {
    $node.removeEventListener(eventName, $node._idom.events[eventName]);

    $node._idom.events[eventName] = eventListener;
  }

  $node.addEventListener(eventName, eventListener);
};

export const applyAttrs = ($node, attrs) => {
  for (const key in attrs) {
    if (typeof attrs[key] === 'function') {
      eventListener($node, key, attrs[key]);
    } else {
      setAttrs($node, key, attrs[key]);
    }
  }
};
```

core.js

```
import { applyAttrs } from './attrs';
import { getElement, removeExcessChildren } from './element';

let currentNode = null;
let currentParent = null;

export const elementOpen = (tag, key = Date.now(), attrs) => {
  const { $element, exists } = getElement(currentParent, tag, key);

  applyAttrs($element, attrs);

  if (exists) {
    // remove existing $element and append $element, to save elements order
    $element.parentNode.removeChild($element);
  }
```

```
    currentParent.append($element);

    if (currentParent._idom && !currentParent._idom.childrenKeys.includes(key)) {
        currentParent._idom.childrenKeys.push(key);
    }

    currentParent = $element;
    currentNode = $element;

    return $element;
};

export const elementClose = () => {
    currentParent = currentNode.parentNode;
    currentNode = currentNode.parentNode;

    return currentNode;
};

export const patch = ($parent, patcher) => {
    currentParent = $parent;

    if (currentParent._idom) {
        currentParent._idom.childrenKeys = [];
    }

    patcher();

    removeExcessChildren(currentParent);
};

export const text = (value) => {
    if (currentParent && currentParent.textContent !== value) {
        currentParent.textContent = value;
    }
};
```

element.js

```
const createNewElement = (tag, key) => {
    const $element = document.createElement(tag);
    if (key) {
        $element._idom = { key, events: {}, childrenKeys: [] };
    }
    return $element;
```

```
};

const getElementByKey = ($parent, key) => Array
   .from($parent.children)
   .find(child => child._idom && child._idom.key && child._idom.key === key);

export const getElement = ($parent, tag, key) => {
   if ($parent && $parent.children.length) {
      const $element = getElementByKey($parent, key);

      if ($element) {
         return { $element, exists: true };
      }
   }

   return {
      $element: createNewElement(tag, key),
      exists: false
   };
};

export const removeExcessChildren = $parent => {
   const children = Array.from($parent.children);

   if ($parent._idom && children.length > $parent._idom.childrenKeys.length) {
      children.forEach(child => {
         if (!$parent._idom.childrenKeys.includes(child._idom.key)) {
            $parent.removeChild(child);
         }
      });
   }
};
```

index.js

```
export { elementOpen, elementClose, patch, text } from './core';
```

diff.js

```
import {render} from './render';

const zip = (x, y) => {
  const z = [];
  const max = Math.max(x.length, y.length);

  for (let i = 0; i < max; i++) {
    z.push([x[i], y[i]]);
  }

  return z;
};

const diffAttrs = (oldAttributes, newAttributes) => {
  const patchers = [];

  // set new attributes
  for (const [key, value] of Object.entries(newAttributes)) {
    patchers.push(node => {
      if (typeof value === 'function') {
        if (key in oldAttributes) {
          node.removeEventListener(key, value);
        }
        node.addEventListener(key, value);
      } else {
        node.setAttribute(key, value);
      }
      return node;
    });
  }

  // remove old attributes
  for (const key in oldAttributes) {
    if (!(key in newAttributes)) {
      patchers.push(node => {
        node.removeAttribute(key);
        return node;
      });
    }
  }

  return node => {
    for (const patch of patchers) {
```

```
      patch(node);
    }
  };
};

const diffChildren = (oldChildren, newChildren) => {
  const childPatchers = [];

  oldChildren.forEach((oldVChild, i) => {
    childPatchers.push(diff(oldVChild, newChildren[i]));
  });

  const newPatchers = [];

  for (const child of newChildren.slice(oldChildren.length)) {
    newPatchers.push(element => {
      element.appendChild(render(child));

      return element;
    });
  }

  return parent => {
    for (const [patcher, child] of zip(childPatchers, parent.childNodes)) {
      patcher(child);
    }

    for (const patcher of newPatchers) {
      patcher(parent);
    }

    return parent;
  };
};

export const diff = (oldElement, newElement) => {
  if (newElement === undefined) {
    return node => {
      node.remove();

      return undefined;
    };
  }
```

```
  if (typeof oldElement === 'string' ||
    typeof newElement === 'string') {

    if (oldElement !== newElement) {
      return node => {
        const newNode = render(newElement);

        node.replaceWith(newNode);

        return newNode;
      };

    } else {

      return () => {
        return undefined;
      };

    }
  }

  if (oldNode.tagName !== newNode.tagName) {
    return node => {
      const newNode = render(newNode);

      node.replaceWith(newNode);

      return newNode;
    };
  }

  const patchAttributes = diffAttrs(oldNode.attrs, newNode.attrs);
  const patchChildren = diffChildren(oldNode.children, newNode.children);

  return node => {
    patchAttributes(node);

    patchChildren(node);

    return node;
  };
};
```

element.js

```
export const createElement = (
```

```
  tagName,
  {
    attrs = {},
    children = []
  } = {}
) => ({
  tagName,
  attrs,
  children,
});
```

mount.js

```
export const mount = (
  node, target
) => {
  target.replaceWith(node);

  return node;
};
```

render.js

```
const renderElem = (options) => {
  const el = document.createElement(options.tagName);

  const entries = Object.entries(options.attrs);

  // set attributes
  for (const [key, value] of entries) {
    if (typeof value === 'function') {
      el.addEventListener(key, value);
    } else {
      el.setAttribute(key, value);
    }
  }

  // set children
  for (const child of options.children) {
    const $child = render(child);
    el.appendChild($child);
  }

  return el;
};

export const render = element => {
```

```
  if (typeof element === 'string') {
    return document.createTextNode(element);
  }

  return renderElem(element);
};
```

index.js

```
export {createElement} from './element';
export {mount} from './mount';
export {render}  from './render';
export {diff}  from './diff';
```

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО”
Кафедра автоматизованих систем обробки інформації і управління

УЗГОДЖЕНО

Керівник проєкту

_____ *Олена Гавриленко*

(підпис) (вл. ім’я, прізвище)

“13” квітня 2020 р.

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ *Олександр ПАВЛОВ*

(підпис) (вл. ім’я, прізвище)

“14” квітня 2020 р.

Комплекс задач аналізу моделей для побудови
Document Object Model

ТЕХНІЧНЕ ЗАВДАННЯ

Шифр *ДП 6109.01.000 ТЗ*

на 10 сторінках

Київ – 2020 року

ЗМІСТ

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Повне найменування системи та її умовне позначення

Повна назва системи: «Віртуальна та інкрементальна моделі для побудови Document Object Model».

1.2 Найменування організації-замовника та організації-учасника робіт

Замовником є кафедра автоматизованих систем обробки інформації та управління Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського» (далі за текстом — Замовник). Адреса замовника: м. Київ, п. Перемоги 37, 18 корпус ФІОТ.

Розробник сервісу — студент групи ІС-з61 кафедри автоматизованих систем обробки інформації та управління Національного технічного університету України “Київський політехнічний інститут ім. Ігоря Сікорського" Загоровський Олександр Анатолійович.

1.3 Перелік документів, на підставі яких створюється система

При розробці системи і створення проектно-експлуатаційної документації Виконавець повинен керуватися вимогами наступних нормативних документів:

- ДСТУ 19.201-78. Технічне завдання. Вимоги до змісту і оформлення;
- ДСТУ 34.601-90. Комплекс стандартів на автоматизовані системи. Автоматизовані системи. Стадії створення;
- ДСТУ 34.201-89. Інформаційні технології. Комплекс стандартів на автоматизовані системи. Види, комплектність і позначення документів при створенні автоматизованих систем.

1.4 Планові терміни початку і закінчення роботи зі створення системи

Плановий термін початку роботи над створенням системи – 13 квітня 2020 року.

Плановий термін по закінченню роботи над системою – 2 червня 2020 року.

2 ПРИЗНАЧЕННЯ І ЦІЛІ СТВОРЕННЯ КОМПЛЕКСУ ЗАДАЧ

2.1 Призначення комплексу задач

Комплекс задач аналізу моделей для побудови Document Object Model призначений для підвищення ефективності роботи вебзастосунків, а саме роботи з DOM.

Ефективна робота з DOM у вебзастосунках дає змогу будувати складні інтерфейси з великою кількістю одночасно відображених елементів на сторінці та при оновленні цих елементів – зберегти стабільну кількість кадрів відмальовування у браузері. Це є однією із ключових складових користувацького досвіду при використанні вебзастосунків.

2.2 Цілі та задачі створення комплексу задач

Цілю створення комплексу задач аналізу моделей для побудови Document Object Model є спрощення вибору моделі DOM при розробці будь-якого вебзастосунку в залежності від вимог до нього.

Для досягнення поставленої цілі необхідно розробити власні реалізації віртуальної та інкрементальної моделей DOM та проаналізувати їхню роботу за допомогою наступного комплексу задач:

1) задачі аналізу швидкості побудови та зміни DOM:
   - створення таблиці що містить 1000 рядків;
   - створення таблиці що містить 10000 рядків;
   - оновлення таблиці що містить 1000 рядків;
   - видалення таблиці що містить 10000 рядків;
   - оновлення тексту кожного 10-го рядка для таблиці що містить 10000 рядків;
   - видалення одного рядка з таблиці що містить 1000 рядків у відповідь на натискання на рядок;
   - заміна місцями 2 рядків в таблиці що містить 1000 рядків;
   - видалення одного рядка з таблиці що містить 1000 рядків;

- додавання 1000 рядків до таблиці що містить 10000 рядків.

2) задачі оцінки кількості використаної пам’яті про роботі з DOM:
   - створення таблиці що містить 1000 рядків;
   - оновлення кожного 10 рядка таблиці що містить 1000 рядків;
   - додавання 1000 рядків до існуючої таблиці;
   - створення та видалення таблиці що містить 1000 рядків.

3 ХАРАКТЕРИСТИКА ОБ'ЄКТА АВТОМАТИЗАЦІЇ

Об'єктом автоматизації є процес побудови та зміни Document Object Model у браузері.

Процес побудови та зміни Document Object Model включає у себе:

- створення елементів інтерфейсу за допомогою Document API;
- оновлення елементів інтерфейсу за допомогою Document API;
- оптимізація роботи з DOM за допомогою запису та аналізу Snapshot у Chrome DevTools;

4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вимоги до функціональних характеристик

- Реалізації віртуальної та інкрементальної моделей мають запускатися та коректно працювати на всіх сучасних операційних системах та в основних підтримуваних версіях сучасних браузерів.
- Архітектура реалізацій віртуальної та інкрементальної моделей мають бути гнучкими та придатними для розширення функціоналу.

4.2 Вимоги до надійності

Реалізації віртуальної та інкрементальної моделей повинні адекватно реагувати на помилки застосування та видавати відповідні повідомленні розробнику що їх використовує.

4.3 Умови експлуатації

Для адекватної роботи реалізацій необхідний пристрій з платформою, яка відповідає вимогам зазначеним в розділі 4.4.

Усі користувачі системи повинні дотримуватися правил експлуатації електронної обчислювальної техніки.

4.4 Вимоги до складу і параметрів технічних засобів

Для правильної роботи реалізацій віртуальної та інкрементальної моделей до складу технічних засобів повинні входити:

Комп’ютер з конфігурацією не гірше:

- Процессор: Dual Core from Intel or AMD at 2.8 GHz;
- Об’єм оперативної пам’яті: 4 GB RAM;
- Відеокарта: Intel HD 3000;
- Storage: 450 MB available space.

Додатково має бути встановлене таке програмне забезпечення:

- Операційна система Windows 7 або новіша;
- DirectX: Version 9.0c.

Комп’ютерна периферія, до складу якої входить:

- Монітор;
- Клавіатура;
- Мишка.

5 СТАДІЇ І ЕТАПИ РОЗРОБКИ

У таблиці 5.1 наведено календарний план робіт та терміни їх виконання.

Таблиця 5.1 – Календарний план виконання робіт

№ з/п	Назва етапів створення продукту	Термін виконання етапу	Результат виконання
1.	Підготовка технічного завдання на розробку програмного продукту	13.04.20	Виконано
2.	Розробка сценарію роботи	17.04.20	Виконано
3.	Технічне проектування – функціональність, модулі, задачі, цілі тощо	23.04.20	Виконано
4.	Узгодження з керівником інтерфейсу користувача	30.05.20	Виконано
5.	Алгоритмізація задачі	05.05.20	Виконано
6.	Розробка інформаційного забезпечення	22.05.20	Виконано
7.	Розробка програмного забезпечення	07.05.20	Виконано
8.	Налагодження програми	08.06.20	Виконано
9.	Тестування програми	01.06.20	Виконано
10.	Здача готового програмного продукту замовнику	08.06.20	Виконано

6 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ СИСТЕМИ

6.1 Види випробувань

Види випробувань узгоджуються із замовником до проведення випробувань. Здача - прийом робіт виконується поетапно на комп'ютерах замовника в аудиторіях кафедри АСОІУ у відповідності з робочою програмою та календарним планом.

Усі програмні продукти, що створюються в рамках даної системи передаються замовнику як у вигляді готових модулів, так і у вигляді вихідних кодів, представлених в електронній формі.

Графічний матеріал
до дипломного проєкту

на тему: Комплекс задач аналізу моделей для побудови Document Object Model

Київ – 2020 року

					ДП 6109.01.000 ССВ		
Зм.	Арк.	№ документа	Підпис	Дата			
Розробив		Загоровський О.А.			Схема структурна варіантів використань	Література	Маса / Масштаб
Перевірив		Гавриленко О.В.				Аркуш 1	Аркушів 1
Т. кон.					Комплекс задач аналізу моделей для побудови Document Object Model	КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-з61	
Н. кон.		Тєлишева Т.О.					
Затвердив		Гавриленко О.В.					

					ДП 6109.02.000 ССС		
Зм.	Арк.	№ документа	Підпис	Дата	Схема структурна станів системи	Література	Маса / Масштаб
Розробив		Загоровський О.А.					
						Аркуш 1	Аркушів 1
Перевірив		Гавриленко О.В.					
Т. кон.					Комплекс задач аналізу моделей для побудови Document Object Model	КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-з61	
Н. кон.		Тєлишева Т.О.					
Затвердив		Гавриленко О.В.					

ДП 6109.03.000 ССК

					ДП 6109.03.000 ССК		
Зм.	Арк.	№ документа	Підпис	Дата	Схема структурна класів програмного забезпечення	Література	Маса / Масштаб
Розробив		Загоровський О.А.					
						Аркуш 1	Аркушів 2
Перевірив		Гавриленко О.В.					
Т. кон.					Комплекс задач аналізу моделей для побудови Document Object Model	КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-з61	
Н. кон.		Тєлишева Т.О.					
Затвердив		Гавриленко О.В.					

ДП 6109.03.000 ССК

					ДП 6109.03.000 ССК		
Зм.	Арк.	№ документа	Підпис	Дата	Схема структурна класів програмного забезпечення	Література	Маса / Масштаб
Розробив		Загоровський О.А.					
						Аркуш 2	Аркушів 2
Перевірив		Гавриленко О.В.					
Т. кон.					Комплекс задач аналізу моделей для побудови Document Object Model	КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-361	
Н. кон.		Тєлишева Т.О.					
Затвердив		Гавриленко О.В.					

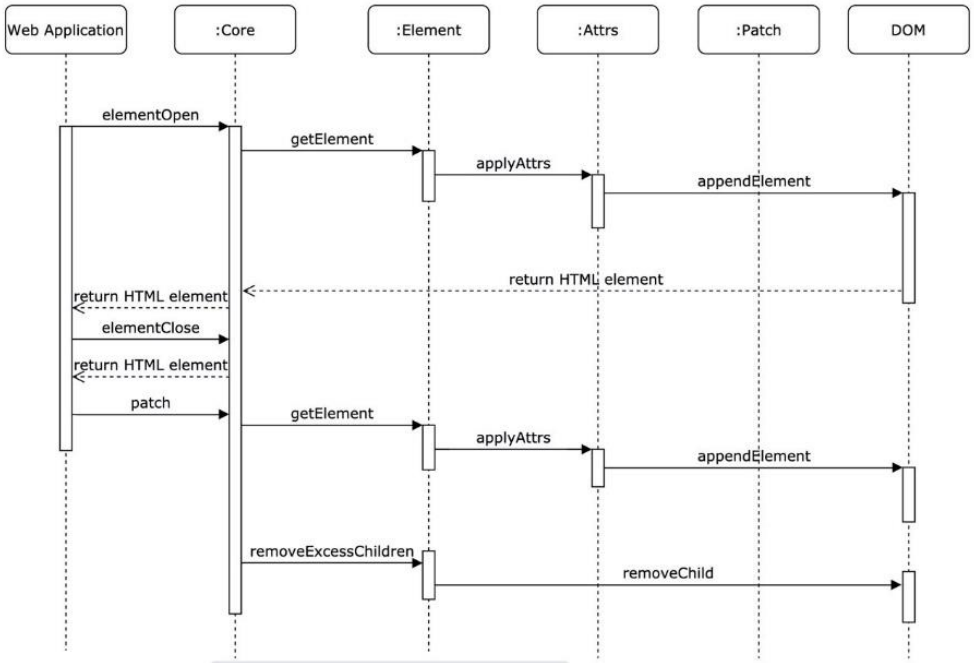


Схема структурна послідовності процесу побудови та зміни IDOM

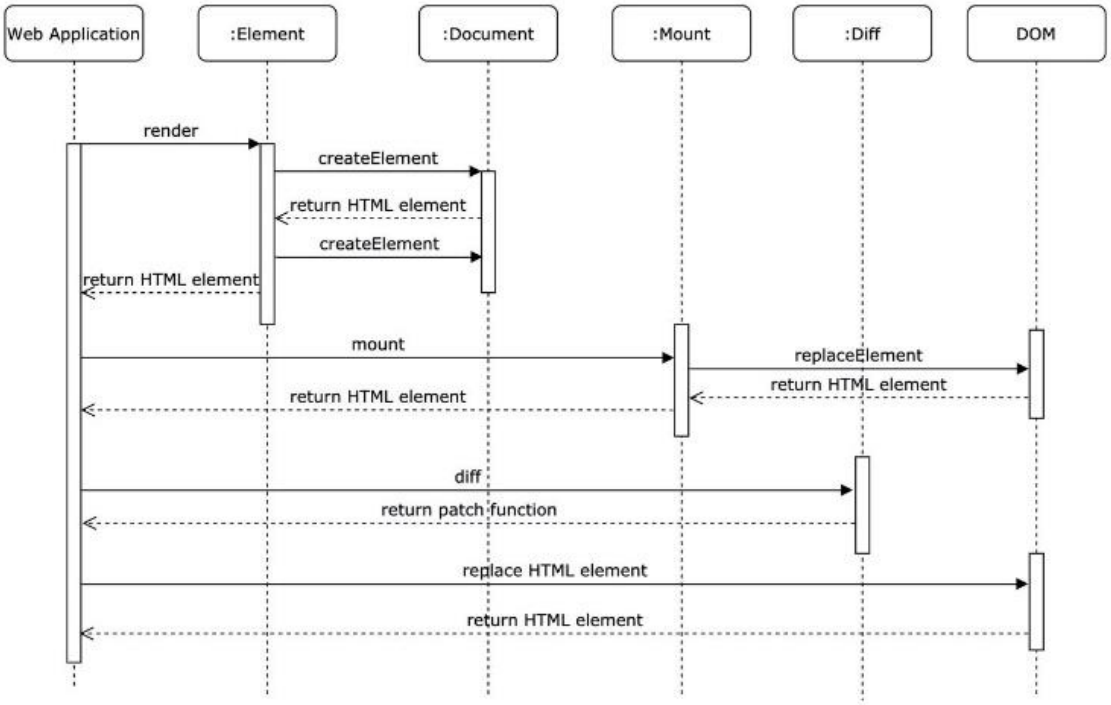
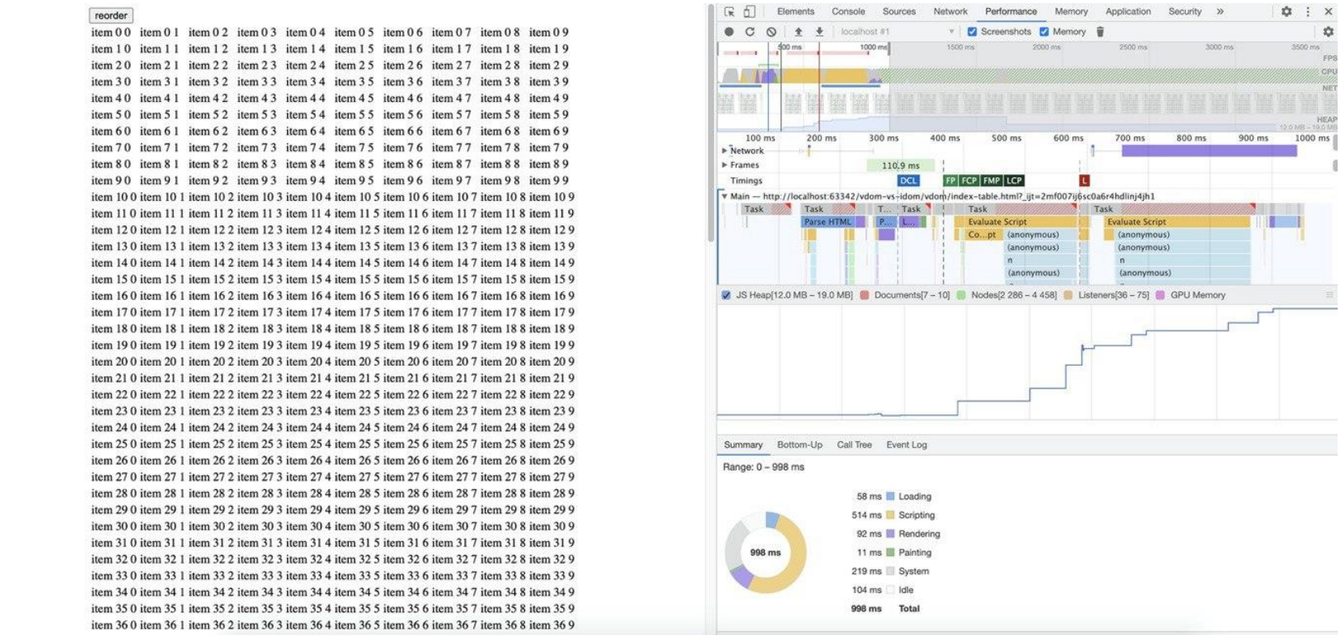
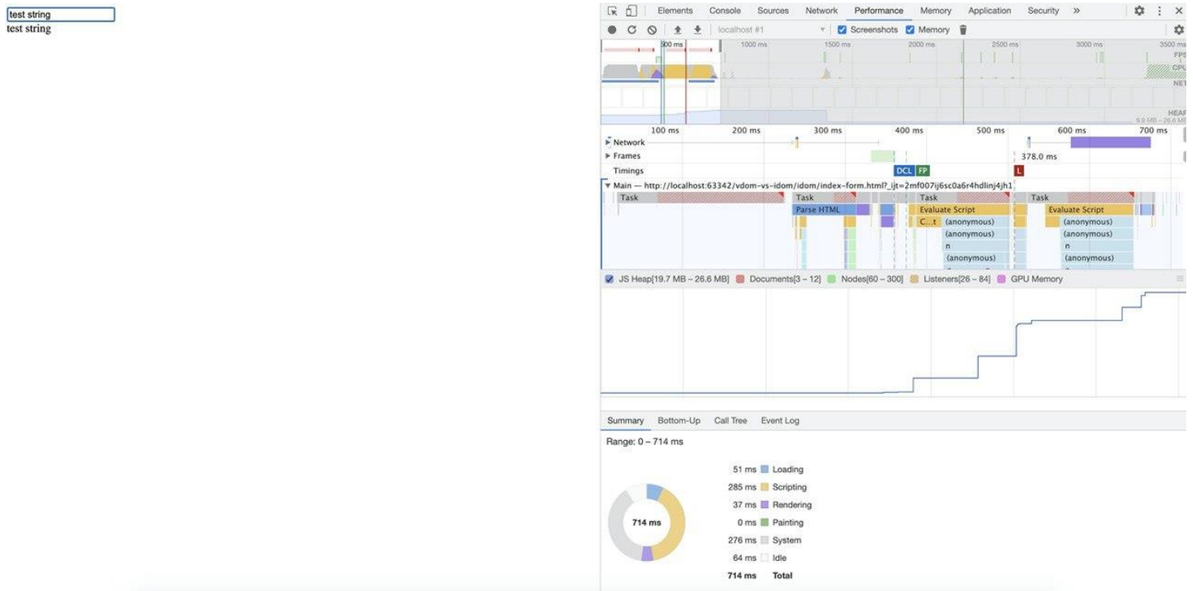


Схема структурна послідовності процесу побудови та зміни VDOM

					ДП 6109.04.000 ССП		
Зм.	Арк.	№ документа	Підпис	Дата	Схема структурна послідовностей	Література	Маса / Масштаб
Розробив		Загоровський О.А.					
						Аркуш 1	Аркушів 1
Перевірив		Гавриленко О.В.					
Т. кон.					Комплекс задач аналізу моделей для побудови Document Object Model	КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-з61	
Н. кон.		Тєлишева Т.О.					
Затвердив		Гавриленко О.В.					



Екранна форма сторінки побудованої за допомогою моделі IDOM для виміру швидкості побудови таблиці що складається з 1000 елементів



Екранна форма сторінки побудованої за допомогою моделі VDOM для виміру швидкості зміни DOM елементу

					ДП 6109.05.000 КЕ	
Зм.	Арк.	№ документа	Підпис	Дата	Креслення вигляду екранних форм	Літера / Маса / Масштаб
Розробив		Загоровський О.А.				
Перевірив		Гавриленко О.В.				Аркуш 1 / Аркушів 1
Т. кон.						
Н. кон.		Тєлишева Т.О.			Комплекс задач аналізу моделей для побудови Document Object Model	КПІ ім. Ігоря Сікорського Каф. АСОІУ Гр. ІС-361
Затвердив		Гавриленко О.В.				

UNICHECK

Власник документу:
Попенко Володимир Дмитрович

ID перевірки:
1004021104

Дата перевірки:
13.06.2020 21:15:47 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
13.06.2020 21:17:23 EEST

ID користувача:
77149

Назва документу: +Zagorovskij_isz61

ID файлу: 1004027869 Кількість сторінок: 51 Кількість слів: 9119 Кількість символів: 66765 Розмір файлу: 88.30 KB

5.49% Схожість

Найбільша схожість: 1.22% з джерело https://juejin.im/post/5c052f95e51d4523d51c8300

4.9% Схожість з Інтернет джерелами	40	Page 53
1.77% Текстові збіги по Бібліотеці акаунту	97	Page 53

0.7% Цитат

Цитати	1	Page 54

Вилучення переліку посилань вимкнено

0% Вилучень

Вилучений текст відсутній

Підміна символів

Не знайдено замінених символів